FUNDAMENTOS DE LOS ORDENADORES

VOLUMEN II.

ELEMENTOS DE METATEORIA

Por:
Gregorio Fernández Fernández
Fernando Sáez Vacas.

Segunda edición

Edita e imprime:
Deptº de Publicaciones Escuela
T.S. Ingenieros Telecomunicación.
Ciudad Universitaria, s/n.
MADRID. - 3

I.S.B.N. 84-7402-076-X
Depósito Legal M-24822

Cubierta del libro: DETALLE DE UNO DE LOS ESQUEMAS DE AUTOMATAS CALCULADORES DE TORRES QUEVEDO, PUBLICADOS EN 1.913. (Citados y reproducidos en J.G. Santesmases, "Switching research in Spain". Proceedings of an international symposium on the theory of switching held on the Computation Laboratory, Harvard University, Cambridge, Mass., 2-5 abril 1.957 y Ch. & R. Eames, "A Computer perspective", Harvard University Press, Cambridge, Mass., 1.973).

Diseño gráfico de la cubierta: P. Lara.

LEONARDO TORRES QUEVEDO (1.852-1.936) ingeniero español, precursor de la cibernética, de la automática y de la informática, contestaba así en 1.915 a una de las preguntas que le formulaba la revista Scientific American:

"The ancient automatons... imitate the appearance and movements of living beings, but this has not much practical interest, and what is wanted is a class of apparatus which leaves out the mere visible gestures of man and attempts to accomplish the results which a living person obtains, thus replacing a man by a machine". (Citado en Ch. & R. Eames, ut supra p.67).

INDICE VOLUMEN II

PROLOGO A LA PRIMERA EDICION

Este libro ha sido escrito con el propósito de servir de texto a los estudiantes de la asignatura FUNDAMENTOS DE ORDENADORES en tercer curso del Plan de Estudios llamado "1.964 modificado" de la Escuela Técnica Superior de Ingenieros de Telecomunicación de Madrid, curso que hizo su debut en el período académico 1.977-78. De conformidad con este propósito se ha sacrificado toda tendencia al lucimiento o a la erudición en aras de un mejor rendimiento didáctico de los conceptos básicos.

Una rápida ojeada al variado abanico de temas que componen los dos volúmenes le dirá al lector advertido por qué hemos decidido escribir un texto, siendo así que los contenidos que en él se iban a incluir no tenían posibilidad alguna de ser originales, salvo tal vez en la presentación. El comprenderá ipso facto que tales contenidos habrían obligado al alumno a manejar necesariamente no menos de seis libros distintos, no siempre en castellano y, de toda evidencia, no al nivel, introductorio aunque no superficial, que se requería.

Así pues, la variedad del programa de la asignatura, un tanto incongruente éste (por razones complejas) con el título de la misma, ha sido factor decisivo para la creación del texto y ha condicionado su realización concreta y los resultados de la misma. Pero antes de describir la estructura del texto, hay que aclarar que, por lo que respecta al dicho programa, este texto no lo cubre totalmente sino que está prevista su complementación con otros temas, que la fluctuante duración de los cursos académicos en nuestro país hará que se pueda o no se pueda (o se pueda sólo en parte, si aplicamos al razonamiento los principios de la lógica borrosa, Vol. II, Tema 1, Capítulo 8, apartado 3) llevar a cabo.

Sería muy largo, y seguramente fuera de lugar, justificar la elección de la composición de un programa que, por lo que conozco de aquí y de allá, no posee la más ligera semejanza con el de ninguna otra asignatura de Europa y de América. Aún no entrando en ello, pido que se nos crea que nuestra meta no era conseguir la originalidad en la mezcla para compensar la cotidianeidad de los componentes, ni dejarnos conducir por gustos particulares.

Para quien no tenga por qué saberlo, le diré que esta asignatura es impartida a estudiantes sin conocimientos previos de electrónica digital y es la única asignatura obligatoria de informática (¿debería llamarse Fundamentos de Informática?) en una carrera predominantemente tecnológica, cuya Estrella de Oriente es el binomio electrónica - información. Su paño se ha tejido con materiales duraderos, si bien una elemental lógica aconsejaba añadir en el traje (la asignatura y, por ende, el texto) ciertos detalles que parecían demandar las características del cliente (futuro ingeniero de telecomunicación) y la misma actualidad. Resumiendo, el libro que el amable lector tiene en sus manos ha sido construido bajo un criterio de maximización de la tasa de invariancia, dentro de los condicionantes generales que acabamos de mencionar; criterio cuyo carácter utópico no descartamos del todo si, tal como enunció Heráclito, "nada es permanente excepto el cambio" y éste, en el tiempo que vivimos, es tan acelerado que nos amenaza con el shock (Toffler).

Después de largas, aunque amistosas discusiones, con el grupo que más de cerca ha vivido la primera andadura de este texto en su versión casera, se ha llegado al acuerdo de organizarlo en el orden seguiente:

VOLUMEN I. INTRODUCCION AL HARDWARE Y AL SOFTWARE

I.1. Representación y codificación de la información.

I.2. Ordenador EIT-2: Introducción a los lenguajes de máquina y ensambladores.

I.3. Estructura y funcionamiento de los ordenadores.

I.4. Elementos de tecnología.

VOLUMEN II. ELEMENTOS DE METATEORIA

II.1. Lógica.

II.2. Autómatas.

II.3. Algoritmos, programación estructurada y máquinas de Turing.

II.4. Lenguajes.

La secuencia didáctica no tiene por qué coincidir con la secuencia de temas en el documento. Si bien es verdad que el primer volumen, con los complementos que se estimase pertinentes, podría servir para un curso de Programación y Estructura de Ordenadores[1] *y el segundo, también complementado, para un curso de Introducción a lo que algunos llaman Informática Teórica*[1]*, el conjunto de los dos volúmenes puede utilizarse de*

---

1. *Un sistema de clasificación de los temas de Ciencias de la Computación para fines bibliográficos (ver Communications of the A.C.M., Junio 1.975, Vol. 18 N° 6) nos dice que los temas del primer volumen caen dentro de los apartados 6, hardware, y 7, software, y la práctica totalidad del segundo volumen en el apartado 5.2, metateoría, del conjunto 5, matemáticas de la computación.*

   En el segundo volumen se ha atemperado, en la medida de lo posible, la habitual carga formal de sus temas mediante el recurso a ciertos ejemplos y al aligeramiento del aparato matemático.

muy diversas maneras.

Por lo que a nosotros respecta, y para el curso 1.978-1.979, se ha escogido la manera:

I.1 → I.2 → II.1 → II.2 → I.3 → I.4 → II.3 → II.4

guiados por el sabio principio de "dar una de cal y una de arena", sin que seamos capaces de precisar en términos absolutos cuál es de cal y cuál es de arena, ya que la experiencia nos ha demostrado que un tema es cal y es arena al mismo tiempo, dividiendo, pues, al alumnado en dos partes. Me atrevo a asegurar que con la secuencia anterior siempre daremos una de cal y una de arena (más exactamente "dos de cal y dos de arena") ya que, según hemos creido observar, para uno cualquiera de nuestros alumnos, si el volumen I es cal/arena el volumen II es arena/cal.

Para llegar a cubrir el objetivo de la asignatura serían necesarios los que pudieramos llamar volúmenes cero y tres, compuestos por contenidos menos invariantes, que no hemos escrito ni pensamos escribir. El volumen cero precede inexcusablemente a todo lo demás. Se trata de que, con él, el alumno conozca las generalidades de la informática a un nivel introductorio. ¿Qué son, cómo han nacido y evolucionado los ordenadores?. ¿Cómo se programan?. ¿Qué tipos de lenguajes utilizan?. ¿Qué clases de máquinas periféricas se conectan con ellos? ¿Qué es un sistema operativo y un compilador?. ¿Para qué se utilizan los ordenadores?. A un alumno universitario se le puede pedir que lea un librito de unas cien páginas donde se desarrollen estas cuestiones. Durante el curso 1.977-78 nosotros hemos recomendado el libro "Fronteras de la Ciencia. Vol. 1 - Bioquímica, ecología, informática", Biblioteca Educación Permanente. Serie Aula Abierta. U.N.E.D. 1.977.

El volumen III tendrá por objeto describir con algún detalle diversas formas o áreas de aplicación de los ordenadores y el impacto de la informática en nuestra vida actual y futura (al menos, inmediata). La extensión y contenido de este volumen variará según la duración del curso y podrán intercalarse sus temas con los de la secuencia más arriba reseñada (o la que escoja el profesor o el lector, en su caso), o bien dejarlo todo para el final. Mas, aun cuando circunstancias de tiempo impidieran desarrollar formalmente, -quiero decir en clase (esto va dirigido a los profesores que vieran en nuestro texto su texto de curso)-, el volumen III, éste deberá recomendarse a los alumnos como lectura autónoma bajo prescripción facultativa si quieren completar su periplo iniciatorio en el campo de la informática. En la fecha en que redacto este prólogo no se me ocurre mejor consejo para materializar el volumen III que la obra de M.A. Arbib "Ordenadores y Sociedad Cibernética" (sus capítulos 1 y 2 sirven, además, de volumen cero), cuya traducción al castellano y supervisión he tenido el placer de recomendar y realizar, respectivamente, para la editorial AC. Es un libro muy bien escrito, didáctico (se presta muy bien al régimen de lectura autónoma) y posee plena actualidad.

Esta primera edición es el resultado de un trabajo colectivo y urgente, circunstancias que se reflejan inexorablemente en la materialidad de la obra y que, a mi modo de ver, si dejan a ésta muy lejos de la perfección formal, le añaden un valor de documento. Fue escrita en dos versiones, la primera en dura carrera con la impartición de las clases correspondientes[1], de Septiembre de 1.977 a Junio de 1.978, que ha ido apareciendo en fascículos[2] en dicho período. La segunda, hasta

1. El curso tuvo que iniciarse sin posibilidad alguna de prepararlo previamente y sin textos.
2. Exceptuando el 2°tema del 1er. volumen, que es anterior y está publicado ya en "Programación en lenguaje ensamblador. Organización de máquinas. Sistema operativo". Coordinador F. Sáez Vacas. E.T.S.I.I.M. 1a. edición 1.977.

Septiembre de 1.978, con prisas y vacaciones, para la edición que nos ocupa.

Diferentes estilos de redacción, diversas formas de estructurar los contenidos y hasta de utilizar la bibliografía, distintos grados de autoexigencia en los autores, dibujos de tantas y tan variadas manos (y ninguna profesional) son aspectos, entre otros, conectados directamente a la escasez de tiempo que hemos podido destinar a esta tarea. Si tomamos por cierto aquello de que cuando una cosa se pule y se pule pierde espontaneidad, tengo para mí que este libro ha de ser uno de los más espontáneos que nunca se hayan escrito sobre informática. Pero, si pese a todo no fuera uno de los más espontáneos[1], sin duda seguiría siendo uno de los que con más entusiasmo se han pergeñado, a través de un proceso especialmente indicado para elaborar textos de enseñanza (y más exactamente, libros de trabajo) bajo circunstancias poco propicias.

Experimento una singular alegría al citar y agradecer a todos los que nos han ayudado. A los autores de cada parte no los menciono sino globalmente ya que su nombre figura siempre en cabeza de su bloque de páginas, siendo responsables de las mismas, en cumplimiento del encargo que Gregorio Fernández[2] y F. Sáez Vacas, diseñadores y coordinadores del curso y del texto, les hicieron. Cito sólo explícitamente a Juan Quemada que, colaborador en otra Cátedra y por tanto sin obligación ninguna, nos ha echado una mano en nombre de la amistad. Mención especial merece un grupo de alumnos, -no reseño sus nombres por auténtico temor a olvidarme de alguno- que, sin esperar recompensa, han dibujado esquemas, gráficos y organigramas,

---

1. En todo caso, parece que va ser difícil batirlo como uno de los de prólogo más extenso.

2. Que ha sido, con mucho, el autor de mayor contribución.

han planteado sugerencias y detectado erratas. No creo exage-- rar si digo que, gracias a esa ayuda, -poco corriente en la - época universitaria de ahora-, ha sido posible que determina-- dos fascículos salieran del horno más o menos a tiempo. Angelines Villar ha estado sin fallos unida a la máquina y la máquina a ella y únicamente el excusable deseo de asistir a su propio parto ha hecho que cediera, lo que se dice en el minuto - último, su máquina a Juana Burillo, para que ésta mecanografiase el último fascículo. Gracias sean dadas a ambas, y a Angelines de manera particular. Por último, Darío Maravall y José María Vela, en trance de incorporarse a nuestro grupo, han con-- tribuido con algunos dibujos para la segunda versión. Resumiendo, un equipo humano estupendo. Y esto último ha sido para mí, de entre todo lo demás, lo mejor.

F. Sáez Vacas
Canto Blanco (Villanueva de la Cañada),
Septiembre 1.978

PROLOGO A LA SEGUNDA EDICION

La primera edición de este libro ha corrido la mejor suerte que cabía desearle: se ha agotado. Y además, ha recibido un premio, el premio Proceso de Datos 1978.

Para preparar una segunda edición nos hemos concentrado básicamente en reducir la plaga que azota de manera secular a este tipo de obras. Me refiero a las erratas, de las que se han corregido todas las que hemos podido encontrar o se nos han señalado.

Aprovechando la ocasión, se han introducido cambios para coordinar algunos conceptos entre el tema 3, ESTRUCTURA y FUNCIONAMIENTO DE LOS ORDENADORES y el tema 4, ELEMENTOS DE TECNOLOGIA del primer volumen. En este último, hemos sustituido el apéndice "Descripción de algunos microprocesadores" por uno nuevo, más sintético, y se le han añadido al tema tres ejercicios resueltos, que ayudarán bastante -según creemos- a contrastar la comprensión del lector acerca del propio tema.

En el segundo volumen, los temas 1, LOGICA y 2, AUTOMATAS se benefician desde ahora con la introducción de algunos nuevos ejemplos resueltos. También se ha incorporado en él el prólogo (los prólogos, porque ahora son dos), ya que nos hemos dado cuenta de que fuera de este centro académico hay no pocas personas que se hacen con este volumen, pero no con el primero, y tienen derecho, si quieren, a conocer la gestación y estructura del conjunto de la obra, así como el sentido de nuestro modesto homenaje a Torres Quevedo, que es la cubierta de ambos volúmenes. En la primera edición, sólo el volumen primero incluía una página de explicación de la cubierta, siendo así que, en último extremo, el contenido del segundo aparece más directamente vinculado con los trabajos de Torres Quevedo. Tal como

queda resuelto este pequeño problema en la segundo edición es más justo para los lectores exclusivos del Volumen II y para - Torres Quevedo, a costa de un incremento de 8-9 páginas para - los lectores de los dos volúmenes, que, estamos seguros, sa- - brán disculparnos por ello.

F. Sáez Vacas
Junio 1.979.

TEMA 1

LOGICA

JUAN QUEMADA VIVES
Y
GREGORIO FERNANDEZ FERNANDEZ

INDICE

Pág.

Pág.

Pág.

Pág.

CAPITULO 1

INTRODUCCION A LA LOGICA MATEMATICA

1. Introducción.

Hay preguntas como ¿qué es la lógica? o ¿cómo pensamos? que muy pocos sabríamos contestar con claridad. Los principios de la lógica los utilizamos a menudo; por ejemplo, cuando en una conversación intentamos no contradecirnos, que en términos lógicos significaría afirmar algo y su contrario a la vez. Los conocemos intuitivamente pero no de una forma sistematizada. Ordenar y clarificar nuestra forma de pensar no es tarea fácil. Aristóteles fue el primero en comenzarla. Su lógica fue una creación de las más acabadas. La agrupó bajo el nombre de "Organon", que significa "Instrumento". Instrumento para el trabajo intelectual quiso decir seguramente. De todas formas la lógica que desarrolló y que continuó inalterada durante la Edad Media no tuvo la sistematización que tiene la lógica actual.

El paso definitivo se dio con la introducción de una simbología matemática. Es entonces cuando aparece lo que llamamos lógica matemática. Los largos tratados de lógica aristotélica de los libros de filosofía son sustituidos por libros concisos y rigurosos llenos de fórmulas matemáticas. Donde antes figuraba una proposición como "Juan rie" ahora figura la variable proposicional "x" que puede en un caso concreto tomar el significado "Juan rie". El contenido es el mismo sólo que la lógica actual tiene un alcance mucho mayor. La nueva simbología permite unos análisis mucho más completos y la ampliación hacia campos que parecian vedados anteriormente como pueden ser las lógicas borrosas o probabilistas que se ven al final del Tema.

El famoso silogismo que aparece en todos los libros de

filosofía: "si Sócrates es hombre y todos los hombres son mortales, entonces Sócrates es mortal", se representaría por $(x \wedge y) \longrightarrow w$, donde x, y, w son las tres proposiciones que aparecen, "$\wedge$" representa la conectiva "y" y " $\longrightarrow$" representa la conectiva "si... entonces....". Como se ve la concisión y simplificación que conseguimos es grandísima. Esta nueva forma de analizar razonamientos y estructuras lógicas no sólo nos dá una mayor concisión sino que al aparecer ahora leyes, deducciones, implicaciones, estudiadas de una forma sistemática se nos ofrece la posibilidad de simularlas facilmente por métodos mecánicos. El cálculo proposicional por ejemplo, tiene aplicaciones en la demostración automática de teoremas.

Podemos dar un paso más, algebraizar la lógica. Existen unas estructuras llamadas Algebras de Boole que se rigen por las mismas leyes que el cálculo proposicional. Esto puede parecer no tener ninguna trascendencia pero, como se verá más adelante, de la simulación de un álgebra de Boole, se llega a la aparición de los circuitos lógicos, que son una parte insustituible de los ordenadores.

2. LENGUAJE OBJETO Y METALENGUAJE.

Cuando nosotros utilizamos un lenguaje, por lo general lo utilizamos en el campo que le es propio, hacemos uso de él para comunicar experiencias, para hablar sobre las cosas que nos rodean pero a veces también, para hablar sobre el lenguaje mismo, sobre sus propiedades o sobre las de otro lenguaje. Esta distinción de funciones se realiza en base a la utilización que se hace de él. Podemos hacer "uso" o hacer "mención" del lenguaje. Podemos decir "Pedro se ríe cuando ve una película de los Hermanos Marx" o "Pedro es un substantivo". En el se

gundo caso también nos expresamos en castellano pero para hablar sobre propiedades del lenguaje. Según la terminología anterior estamos haciendo mención del lenguaje. Así, llamamos - lenguaje objeto a un lenguaje que actúa según su función propia y metalenguaje al lenguaje que se utiliza para hablar del lenguaje objeto.

Aunque en algunos casos hay lenguajes que se crean con el fin de analizar otros lenguajes, es decir, que son metalenguajes siempre, por lo general muchos lenguajes pueden actuar como metalenguajes o como lenguajes objetos. La lógica por -- ejemplo, puede actuar como lenguaje objeto o para expresar propiedades lógicas de otros lenguajes. También existe la posibilidad de utilizar un lenguaje para expresar propiedades de un metalenguaje, es decir, la actuación como metametalenguaje.

En el capítulo sobre lógica proposicional se habla de metalenguajes y metaexpresiones que definimos expresamente - para ayudarnos en la tarea de dar las propiedades de la lógica proposicional. Así se puede obtener una idea más clara de lo - que es un metalenguaje.

3. SINTAXIS, SEMÁNTICA Y PRAGMÁTICA.

Los lenguajes se utilizan para representar simbólicamente las cosas. Cuando se estudian su composición y sus propiedades vemos que constan de unos símbolos que se rigen por determinadas leyes. Estos símbolos se utilizan para representar los - objetos sobre los que trata nuestro lenguaje. Además debe haber alguien que se represente los objetos mediante el lenguaje. El castellano por ejemplo, tiene sus símbolos, substantivos, - verbos, adjetivos, adverbios, artículos....., unas leyes, la - sintaxis, también llamada gramática, y los objetos reales que representa. Los hispanoparlantes lo utilizan para expresarse y

comunicarse entre sí. Como se ve existen tres partes diferentes. Primero están los símbolos y sus leyes, vacíos de contenido. La parte de la lingüística que trata estos problemas es la *sintaxis*. Después viene el segundo paso. Ligar los símbolos con los objetos que representan. La parte de la lingüística que estudia este campo es la *semántica* o *interpretación semántica*. Finalmente, la pragmática se ocupa de las relaciones entre los símbolos y los sujetos que los usan.

Un ejemplo de enunciado perteneciente a la sintaxis es:

> *'algoritmo' es un sustantivo.*

Un ejemplo de enunciado perteneciente a la semántica es:

> *no es cierto que 'algoritmo' derive del griego 'αλγος' ('dolor') y 'αριθμος' ('número') y signifique 'dolor producido por los números'.*

Finalmente,

> *en otro Tema veremos la etimología correcta de la palabra 'algoritmo'.*

es un enunciado perteneciente a la pragmática.

En el primer caso se ha usado 'algoritmo' para relacionarlo con otra expresión ('sustantivo'); en el segundo, para relacionarlo con el objeto que designa (en este caso es más bien el objeto que no designa), y en el tercero, para relacionarlo con quienes lo usan (nosotros).

4. PROPOSICIONES, RAZONAMIENTOS Y CONECTIVAS.

La lógica tradicional hablaba de juicio (acto mental -

mediante el que pensamos) y proposición (lo pensado en el juicio). Enlazando proposiciones mediante conectivas se formaban los razonamientos. Estos razonamientos son el equivalente de las sentencias o enunciados de la lógica actual. El concepto de juicio es imposible de formalizar pero el de proposición y razonamiento enlazan con el de variable proposicional y el de sentencia. Los segundos son la representación simbólica de los primeros. Las conectivas de hoy y las de ayer también presentan una continuidad, las actuales son la representación de las anteriores mediante símbolos abstractos también. La lógica formal se nos presenta por tanto como un paso más de un proceso que se podría llamar el estudio del razonar correctamente. Las proposiciones como "Juan rie", "Pedro va al cine", se representan ahora por "u" o "v"; las conectivas "y", "o", "si y sólo si" por "$\wedge$", "$\vee$", "$\leftrightarrow$" . La representación de los razonamientos se simplifica mucho y se pueden definir con mucha más precisión sus propiedades. Ya vimos cómo el razonamiento clásico que dimos en la introducción se redujo a $(x \wedge y) \longrightarrow w$.

Hay una limitación de la que no hemos hablado todavía. La lógica proposicional representa las proposiciones por letras pero nunca estudia su composición. La encargada de realizar este estudio es la *lógica de predicados*. Para ello define los argumentos y predicados y unos cuantificadores universales que el lector ya conocerá por su uso en matemáticas. Son el "para todos" ($\forall$) y "algunos" ($\exists$). Con estas nuevas herramientas tenemos una capacidad de análisis mucho mayor, capacidad que es utilizada en una tendencia de la matemática actual, llamada la escuela logicista. Aquí no entraremos a estudiar este tipo de lógica, aunque hay que advertir que también encuentra aplicaciones en informática, en ciertos métodos de inteligencia artificial.

5. LAS CONECTIVAS.

Las conectivas son las partículas que se utilizan para enlazar lógicamente dos proposiciones o sentencias. Comprender el significado intuitivo que tiene cada una ayuda a comprender mejor el formalismo que desarrollaremos después. Las variables proposicionales son eso, variables, y por lo tanto su interpretación depende de lo que le asignemos, pero en las conectivas la situación es diferente. Cada una implica un nexo lógico muy concreto entre las proposiciones. Para entender el significado de cada una hay que ligarlo al de interpretación semántica o, dicho de otra forma, el sentido que se da a una conectiva va muy unido al de verdad o falsedad de la expresión resultante. En $x \wedge y$, ("$\wedge$" significa "y") $x \wedge y$ será verdadera si x e y son verdaderas. En cambio $\neg(x \wedge y)$ (" $\neg$ " significa "no") será lo contrario: será verdadera siempre que x e y no sean verdaderas a la vez. Cuando se den las tablas de verdad se darán uno por uno todos los casos posibles para cada conetiva. Estos nexos lógicos también se utilizan en lenguaje ordinario cuando hacemos algún razonamiento.

En la tabla que se dá a continuación tenemos primero el símbolo lógico, luego su denominación y por último la forma que tiene dicho nexo en español normal y corriente. Se dan sólo las más usuales. Se podrían dar más, pero son innecesarias. Para construir un sistema lógico, sólo con "no" e "y" sería suficiente, pero es más intuitivo desarrollar un sistema lógico con todas las conectivas que enumeramos a continuación.

Símbolo	Denominación	Partícula utilizada en lenguaje ordinario
¬	negación	no
∧	conjunción	y
∨	disyunción	o
—>	condicional	si... entonces
<—>	bicondicional	si y sólo si

1°) La negación es la más sencilla de todas. Es la única que se antepone a una expresión únicamente. Las demás siempre enlazan dos expresiones. En álgebra no se utiliza este signo sino una barra encima de la expresión que se quiera negar; es equivalente a éste pero en lógica matemática se representa siempre así. Cuando una expresión es falsa su negación es verdadera y viceversa. Se representa por:

$$\neg x$$

Un ejemplo sería:

"x" significa "2 es un número par"

" ¬x" se lee "2 no es un número par".

2°) La conjunción es otra partícula muy usada en lenguaje común. La expresión resultante es válida si las expresiones que la componen son ambas válidas. Su forma de enlazar dos expresiones es así:

$$x \wedge y$$

Un ejemplo sería:

"x" — "2 es un número primo", "y" — "2 es un número par"

"x ∧ y" se lee entonces" "2 es un número primo y 2 es un - número par".

°) La disyunción es formalmente del mismo tipo que la anterior, une dos expresiones. Para que la expresión resultante sea verdadera deberán ser verdaderas o una u otra o ambas. Su forma de enlazar expresiones es también:

$$x \vee y$$

Ejemplo:

asignando a "x"e "y" el mismo significado que antes,

"x ∨ y" se lee "2 es un número primo o 2 es un número par.

') El condicional relaciona una expresión llamada antecedente con otra llamada consecuente. Si x es el antecedente e y - el consecuente, se escribe:

$$x \longrightarrow y$$

El resultado del condicional es siempre verdadero, salvo en el caso de que sea el antecedente verdadero y el consecuente falso. En el siguiente Capítulo, al hablar de interpretación semántica, analizaremos esta interpretación - del condicional.

Ejemplo:

"x" — "Es un cuadro de Miguel Angel", "y" — "Es muy va-- lioso"

"x —> y" se lee "Si es un cuadro de Miguel Angel, es muy valioso".

5°) El bicondicional se puede asemejar un poco al significado de equivalencia. La expresión resultante es cierta si ambas son ciertas o ambas son falsas. Su forma de enlazar - dos expresiones es:

$$x \longleftrightarrow y$$

Ejemplo:

"x" — "N es un número par", "y" - "N es un múltiplo de 2".

"x $\longleftrightarrow$ y" se lee "N es un número par si y sólo si N es un múltiplo de 2"

Ejemplo:

Para aclarar ideas vamos a estudiar con detenimiento - la siguiente expresión:

$$\neg(x \wedge y)$$

La expresión es sintácticamente correcta como se verá en el próximo capítulo, por lo que es una sentencia, en la - que vamos a analizar su significación intuitiva

"x" significa "los coches vuelan" (es falsa semánticamente).

"y" significa "Pedro es alto" (suponemos que es verdad semánticamente)

se leería:

no es el caso que (los coches vuelan "y" Pedro sea alto)

negación x $\wedge$ y

Como se ve la sentencia es verdadera semánticamente; - si solamente fuese "x $\wedge$ y" ya no lo sería.

CAPITULO 2

LOGICA PROPOSICIONAL

1. SINTAXIS

La lógica formal en su sentido más general es un lenguaje. Como todo lenguaje tiene unos símbolos o signos que representan los objetos que vamos a tratar y unas reglas de formación y de estructuración interna. El estudio riguroso de los signos y las reglas es lo que se denomina sintaxis. Para ayudarnos en la comprensión de la sintaxis utilizamos expresiones y símbolos que no pertenecen a la lógica en si, sino que las definimos aparte. Estos son elementos de un metalenguaje que creamos y que nos van a simplificar mucho ciertas cuestiones. También existen gran cantidad de símbolos matemáticos usuales que tienen una función metalingüística. El mismo idioma castellano es un metalenguaje en un sentido amplio. Pero sólo mencionaremos expresamente los metasímbolos que haya que definir

1.1. El alfabeto

El alfabeto está compuesto por el conjunto de símbolos necesarios para construir el formalismo de la lógica proposicional. En realidad se podría simplificar aún más, pero esta forma de exposición es la más clara. Con solamente dos conectivas se pueden definir las demás; este aspecto se estudiará más a fondo cuando se vea la relación entre lógica y álgebra de Boole.

a) Variables proposicionales.

   Representadas por las letras minúsculas u, v, w,...

b) Símbolos lógicos o conectivas.

   " ¬ " negación

" $\wedge$ " conjunción

" $\vee$ " disyunción

" $\longrightarrow$ " condicional

"$\longleftrightarrow$ " bicondicional

c) Símbolos de puntuación

"(",")"(llamados paréntesis)

1.2. Expresiones, Sentencias y Secuencias de Formación.

En el capítulo anterior se trató el concepto clásico de sentencia de una forma intuitiva. Aquí se va a dar una definición totalmente rigurosa. Para ello vamos a utilizar ciertas expresiones metalingüísticas. En $A_i = (w) \wedge (v)$ encontramos símbolos que no hemos establecido en el alfabeto, concretamente "A_i" y "=". Separadamente no tiene mucho sentido analizarlos. La expresión en su conjunto dice que A_i es equivalente a $(w) \wedge (v)$ y que por tanto podemos representar esta última por A_i.

Otro metasímbolo que vamos a introducir es k, que representa a cualquiera de las conectivas " ", " ", " $\longrightarrow$ ", = "$\longleftrightarrow$ ".

Def. 1.2.1. Llamamos expresión o cadena a toda secuencia finita de símbolos de nuestro alfabeto colocados uno al lado de otro.

Por ejemplo:

x $\vee$ ((w) $\wedge$ (v)),)) ((w ¬ u $\wedge$ ((, ¬(x)

son expresiones.

Def. 1.2.2. Llamamos secuencia de formación a toda se

cuencia finita de expresiones $A_1, A_2 \ldots\ldots A_n$ (cada metasímbolo $A_1, A_2 \ldots\ldots A_n$ representa una expresión) para la que cada A_i - $(1 \leq i \leq n)$ satisface al menos una de las tres condiciones siguientes:

a) A_i es una variable proposicional

b) Existe algún $j < i$ tal que $A_i = \neg(A_j)$

c) Existe algún $j,h < i$ tal que $A_i = (A_j)\ k\ (A_n)$

Def. 1.2.3. Llamamos sentencia a toda expresión que - tiene al menos una secuencia de formación

"Sentencia" equivale a la noción intuitiva de "expre-- sión bien construída". En los tres ejemplos que dimos anteriormente sólo hay dos sentencias:

$\neg(x)$ es sentencia;

x
$\neg(x)$ } es su secuencia de formación.

$(x) \vee ((w) \wedge (v))$ es sentencia también;

w
v
$(w) \wedge (v)$
x
$(x) \vee ((w) \wedge (v))$ } es su secuencia de formación

)) ((w $\neg$ u $\wedge$ ((, en cambio, no es sentencia.

En nuestro formalismo sólo tenemos variables proposicionales y sentencias. Las variables, como les es propio, pue

den tomar valores concretos, que serán proposiciones, expresiones del lenguaje ordinario, expresiones matemáticas y todo lo que pueda ser analizado por la lógica. Las sentencias también se materializan entonces y cobran significado. Pero todo esto no es competencia de la sintaxis y ya se verá con más detalle cuando se traten las interpretaciones.

A partir de ahora se van a suprimir los paréntesis innecesarios, según el criterio habitual seguido en matemáticas, es decir, cuando su supresión no induzca error sobre qué conectiva afecta a cada expresión.

1.3. Axiomas y demostraciones.

El método axiomático en la elaboración de una teoría se basa en construir tal teoria sobre la base de unos axiomas y unas leyes que nos permiten deducir los diferentes teoremas y proposiciones. Los axiomas son verdades indemostrables matemáticamente por definición. Eso no quita que se comprueben empíricamente y en base a que no creen contadicciones en nuestra teoría. Las leyes que se establecen permiten deducir de los axiomas toda la teoría. Un caso muy conocido es el de la geometría, donde de cinco axiomas, los postulados de Enclides, se construye toda la geometría enclidiana. Aquí vamos a seguir este método. Estableceremos unos axiomas que por definición son verdad. La experiencia nos dice que de momento no han creado ninguna contradicción dentro de la lógica, y que nos permiten deducir los teoremas de nuestra teoría. Si alguien encontrase algún fallo en ellos habría que cambiarlos como ha pasado en algunos casos.

Los axiomas de la lógica proposicional son:

1.- $u \longrightarrow (v \longrightarrow u)$

2.- $(u \longrightarrow (u \longrightarrow v)) \longrightarrow (u \longrightarrow v)$

3.- $(u \longrightarrow v) \longrightarrow ((v \longrightarrow w) \longrightarrow (u \longrightarrow w))$

4.- $(u \longleftrightarrow v) \longrightarrow (u \longrightarrow v)$

5.- $(u \longleftrightarrow v) \longrightarrow (v \longrightarrow u)$

6.- $(u \longrightarrow v) \longrightarrow ((v \longrightarrow u) \longrightarrow (u \longleftrightarrow v))$

7.- $(\neg v \longrightarrow \neg u) \longrightarrow (u \longrightarrow v)$

8.- $u \vee v \longleftrightarrow (\neg u \longrightarrow v)$

9.- $u \wedge v \longleftrightarrow (\neg u \vee \neg v)$

Una vez establecidos los axiomas vamos a establecer el concepto de prueba formal, también llamada demostración.

Para ello vamos a definir un concepto previo.

Def. 1.3.1. Dadas una variable proposicional u, y unas sentencias B y A, llamamos substitución de B en A por u, y lo representamos por (B/u)A, a la expresión resultante de sustituir u por B en todos los lugares de A donde aparezca u.

Por inducción se demuestra fácilmente que (B/u)A es una sentencia.

Def. 1.3.2. Llamamos prueba formal a toda secuencia finita de sentencias $A_1, A_2, \ldots, A_n$ para la que cada A_i tal que $1 \leq i \leq n$ satisface al menos una de las tres condiciones siguientes:

a) A_i es un axioma.

b) Existe algún $j < i$, una sentencia B y una variable u tal que $A_i = (B/u)A_j$

c) Existen unos $h, j < i$ tal que $A_h = A_j \longrightarrow A_i$

A toda sentencia A que posea una prueba formal la denominamos demostrable y la representamos por $\vdash A$:

La segunda y la tercera condición se denominan regla de substitución y regla de modus ponens respectivamente.

Para designar al conjunto de todas las sentencias demostrables utilizamos la letra T.

Ejemplo. 1.3.1.:

Vamos a demostrar dos teoremas clásicos de la lógica proposicional. El primero es $\vdash (A \longrightarrow A)$. Intuitivamente todos sabemos que es verdad. Significa que si A es verdad implica que A es verdad. El segundo también es fácil de comprender: $\vdash (A \vee \neg A)$. Quiere decir que o bien A es verdad o bien A no es verdad.

1.	$u \longrightarrow (v \longrightarrow u)$	Axioma 1
2.	$u \longrightarrow (u \longrightarrow u)$	(u/v) 1
3.	$(u \longrightarrow (u \longrightarrow v)) \longrightarrow (u \longrightarrow v)$	Axioma 2
4.	$(u \longrightarrow (u \longrightarrow u)) \longrightarrow (u \longrightarrow u)$	(u/v) 3
5.	$u \longrightarrow u$	modus ponens (2.4)
6.	$(u \longrightarrow v) \longrightarrow (v \longrightarrow u)$	Axioma 5
7.	$(w \longleftrightarrow v) \longrightarrow (v \longrightarrow w)$	(w/u) 6
8.	$(w \longleftrightarrow s) \longrightarrow (s \longrightarrow w)$	(s/v) 7
9.	$(u \vee v \longleftrightarrow s) \longrightarrow (s \longrightarrow u \vee v)$	$(u \vee v/w)$ 8
10.	$(u \vee v \longleftrightarrow (\neg u \longrightarrow v)) \longrightarrow (\neg u \longrightarrow v) \longrightarrow u \vee v)$	$(u \longrightarrow v/s)$ 9
11.	$u \vee v \longleftrightarrow (\neg u \longrightarrow v)$	Axioma 8
12.	$(\neg u \longrightarrow v) \longrightarrow u \vee v$	modus ponens (10,11)
13.	$(\neg u \longrightarrow \neg u) \longrightarrow u \vee \neg u$	(u/v) 12
14.	$(\neg u \longrightarrow \neg u)$	(u/v) 5
15.	$u \vee \neg u$	modus ponens (13,14)

En la sentencia 5 se prueba el primer teorema y en la 15 el segundo (basta sustituir en cada caso A por u).

2. Interpretación Semántica.

2.1. Interpretación de variables y sentencias.

La interpretación de un formalismo es la parte que se ocupa de la relación que tienen los objetos formales definidos sintácticamente por nosotros con los objetos reales que tratan de describir. En lógica proposicional clásica la interpretación sólo se ocupa de si las sentencias son verdaderas o falsas. Otras características que pueda tener una sentencia no importan. El formalismo matemático que vamos a desarrollar aquí es mucho más general y abarca a diferentes formas de interpretación de las sentencias. Permite hacer extensiones de la lógica proposicional clásica a nuevos campos que se están estudiando hoy día, como lógicas borrosas, multivalores, de umbral, probabilistas..... En el último capítulo de este Tema se estudian estos nuevos enfoques con más detenimiento.

Para seguir adelante necesitamos definir unos conceptos previos. Vamos a representar por U a un conjunto de variables proposicionales y por S al conjunto de todas las sentencias que puedan formarse a partir de ellas.

Def. 2.1.1. Llamamos orden de una expresión al número de conectivas que posee.

Al conjunto de todas las sentencias que tengan un orden menor o igual a n lo representamos por S_n.

Def. 2.1.2. Llamamos interpretación del conjunto U al sistema <I,V>, donde I es una aplicación de U sobre V. V es -

un conjunto no vacio, con 5 operaciones (1) definidas en su seno, "¬", "∧", "∨", "—> ", " <—> ". La primera es unitaria y las demás binarias. Estarán definidas por todas partes.

Tal y como definimos la interpretación, ésta solamente abarca a las variables. La forma de interpretar la establecemos nosotros, según se desprende del hecho de no imponer condiciones a I. El asignar valores de verdad o falsedad a las variables se hace según nuestro criterio.

Sobre el conjunto V, tampoco imponemos ninguna restricción fuerte. Solamente tiene que tener definidas en su seno unas operaciones equivalentes a las conectivas. Para interpretar las variables según la lógica proposicional clásica como verdaderas o falsas, solo necesitariamos dos elementos, 0 y 1 por ejemplo. Si la aplicación hace corresponder el 1 a la variable ésta se interpreta como verdadera, si hace corresponder el 0 ésta se interpreta como falsa. Entre el 0 y el 1 habría que establecer operaciones internas. También se podrían utilizar conjuntos V más complejos, por ejemplo uno de 3 posibilidades, validez, incertidumbre y falsedad. También conjuntos infinitos que tuviesen un margen continuo de certidumbre más o menos acentuada. Hechas estas aclaraciones vamos a dar un paso más: Dada una interpretación de variables pasar a interpretar las sentencias. El siguiente teorema nos indica cómo.

Teorema 2.1.3. Dada una interpretación <I,V> existe una extensión I' de la aplicación I, y sólo una, de forma que I' es una aplicación de S en V que satisface las siguientes condiciones:

(1) Operación dentro de un conjunto se define como una aplicación de V^n —> V; si n=1 será unitaria; si n=2, binaria.

a) $I'(\neg A) = \neg I'(A) \quad \forall A \in S$

b) $I'(A \; K \; B) = I'(A) \; K \; I'(B) \quad \forall A,B \in S$

Demostración:

La existencia de I' puede demostrarse por inducción sobre el orden de las sentencias. Para ello, si $A \in S_n$, denotaremos $I'(A)$ por $I_n(A)$. Para $n = 0$ es evidente que existe $I_0(A)$, ya que será precisamente $I_0(A) = I(A)$. Supongamos que existe $I_n(A)$ y que cumple las condiciones a) y b); entonces, definimos la aplicación I_{n+1} así:

$I_{n+1}(A) = I_n(A)$ si $A \in S_n$

$I_{n+1}(A) = \neg I_n(B)$ si $A = \neg B$ y $A \in S_{n+1}$

$I_{n+1}(A) = I_n(B) \wedge I_n(C)$ si $A = B \wedge C$ y $A \in S_{n+1}$

$I_{n+1}(A) = I_n(B) \vee I_n(C)$ si $A = B \vee C$ y $A \in S_{n+1}$

$I_{n+1}(A) = I_n(B) \longrightarrow I_n(C)$ si $A = B \longrightarrow C$ y $A \in S_{n+1}$

$I_{n+1}(A) = I_n(B) \longleftrightarrow I_n(C)$ si $A = B \longleftrightarrow C$ y $A \in S_{n+1}$

así tenemos demostrada la existencia de I_{n+1}, ya que hemos construído una aplicación que cumple las condiciones impuestas.

Además de la existencia de I' hay que demostrar su unicidad. Supongamos que existe otra I" que cumple las condiciones de I'. Cualquier restricción de I" a S_n sería idéntica a la restricción de I' sobre S_n, por la forma en que se ha construído. Es decir que I' es única.

Corolario. Dados un conjunto V y una interpretación I' de S sobre V que cumple las condiciones enunciadas en el teorema 2.1.3, ésta (I') estará univocamente determinada por los valores que corresponden a los elementos de U.

2.2. Interpretación binaria y tablas de verdad.

A partir de ahora, y hasta el Capítulo 8, supondremos que el conjunto V es {0,1} y que las cinco operaciones se definen así:

$$\neg(0) = 1; \quad \neg(1) = 0$$

$$0 \wedge 0 = 0; \quad 0 \wedge 1 = 0; \quad 1 \wedge 0 = 0; \quad 1 \wedge 1 = 1$$

$$0 \vee 0 = 0; \quad 0 \vee 1 = 1; \quad 1 \vee 0 = 1; \quad 1 \vee 1 = 1$$

$$0 \to 0 = 1; \quad 0 \to 1 = 1; \quad 1 \to 0 = 0; \quad 1 \to 1 = 1$$

$$0 \leftrightarrow 0 = 1; \quad 0 \leftrightarrow 1 = 0; \quad 1 \leftrightarrow 0 = 0; \quad 1 \leftrightarrow 1 = 1$$

La lógica así resultante es la lógica binaria clásica, con el valor 0 correspondiente a "falso" y 1 a "verdadero" o "cierto".

Def. 2.2.1. Un sistema de valores de verdad es una interpretación <I,V> en la que V = {0,1} y las cinco operaciones se definen como se acaba de ver.

Por ejemplo, si U = {u,v}, hay cuatro posibles sistemas de valores de verdad:

I_0: u → 0; v → 0

I_1: u → 0; v → 1

I_2: u → 1; v → 0

I_3: u → 1; v → 1

En cada uno de estos sistemas, y de acuerdo con el Teo

rema 2.1.3, cada sentencia formada con u y v tendrá una interpretación. Por ejemplo, la interpretación de $u \wedge v$ en el sistema I_1 será:

$$I'_1(u \wedge v) = I'_1(u) \wedge I'_1(v) = I_1(u) \wedge I_1(v) = 0 \wedge 1 = 0;$$

la interpretación de $\neg(u \vee v) \rightarrow v \wedge (\neg u)$ en el sistema I_2 será:

$$I'_2(\neg(u \vee v) \rightarrow v \wedge (\neg u) = I'_2(\neg(u \vee v)) \rightarrow I'_2(v \wedge (\neg u)) =$$

$$= \neg I'_2(u \vee v) \rightarrow I'_2(v) \wedge I'_2(\neg u) =$$

$$= \neg (I'_2(u) \vee I'_2(v)) \rightarrow I'_2(v) \wedge \neg (I'_2(u) =$$

$$= \neg (I_2(u) \vee I_2(v)) \rightarrow I_2(v) \wedge \neg (I_2(u)) =$$

$$= \neg (1 \vee 0) \ 0 \ \wedge \neg (1) = \neg 1 \rightarrow 0 \wedge 0 = 0 \rightarrow 0 = 1$$

Def. 2.2.2. La tabla de verdad de una sentencia es una representación tabular de las interpretaciones de la sentencia para cada uno de los posibles sistemas de valores de verdad.

A modo de ejemplo, veamos las tablas de verdad de las sentencias formadas por aplicación de las conectivas a variables proposicionales:

	u	v	$\neg u$	$u \wedge v$	$u \vee v$	$u \rightarrow v$	$u \leftrightarrow v$
I_0:	0	0	1	0	0	1	1
I_1:	0	1	1	0	1	1	0
I_3:	1	0	0	0	1	0	0
I_4:	1	1	0	1	1	1	1

Estas cinco tablas deben conocerse de memoria, para poder construir rápidamente a partir de ellas la tabla de verdad

de cualquier sentencia. Por ejemplo, la tabla de verdad del Axioma 3 puede hallarse paso a paso así:

u	v	w	u→v	v→w	u→w	(v→w)→(u→w)	(u→v)→ →((v→w)→(u→w)) (Axioma 3)
0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1
0	1	0	1	0	1	1	1
0	1	1	1	1	1	1	1
1	0	0	0	1	0	0	1
1	0	1	0	1	1	1	1
1	1	0	1	0	0	1	1
1	1	1	1	1	1	1	1

Veamos cómo todo esto puede ayudar al análisis lógico de frases del lenguaje cotidiano. Consideremos la frase

si hay niebla no se ve

Si llamamos u a la proposición "hay niebla" y v a "se ve", la sentencia formalizada es: u→¬v. Las posibles interpretaciones quedan resumidas en la tabla de verdad:

u	v	¬v	u→¬u
0	0	1	1
0	1	0	1
1	0	1	1
1	1	0	0

Esta tabla nos dice que sólo hay un caso en el que la sentencia es falsa: si hay niebla (u=1) y se ve (v=0). En todos los demás casos es verdadera:

* u = 0; v = 0 (puede no haber niebla y no haber visibilidad, por ser de noche).

* u = 0; v = 1 (no hay niebla; se ve)

* u = 1; v = 0 (hay niebla; no se ve)

Este puede ser un buen momento para hacer algunas consideraciones sobre el sentido que, en general, tiene el condicional. Según lo definíamos en el Capítulo anterior, y según queda reflejado en su tabla de verdad, una sentencia de la forma u→v se interpreta como falsa en el caso de que u sea cierto y v falso, y como cierta en todos los demás casos. Analicemos independientemente cada caso:

a) u = 1, v = 1 (antecedente y consecuente ciertos). - Parece evidente que en tal caso el condicional ("si u, entonces v") deberá ser cierto. Así,

'si como mucho, entonces engordo'.

es un condicional cierto en el caso de que tanto el antecedente ("como mucho") como el consecuente ("engordo") lo sean. Pero piénsese que también pueden ser ciertos condicionales en los que no haya una relación causa-efecto entre antecedente y consecuente; así:

'Si Madrid es la capital de España,
entonces París es la capital de Francia'

es un condicional verdadero. Por ello, aunque a veces, en lugar de decir "condicional" se diga "implicación (o "implicación material"), no hay que confundirlo con la llamada "implicación estricta":

'Madrid es la Capital de España' implica que
'París es la Capital de Francia'

es una implicación falsa. La implicación estricta se estudia en otra faceta de la lógica formal, la lógica modal, que no abordaremos aquí.

b) u = 1, v = 0. En este caso parece natural decir que

el condicional es falso. En efecto, decir "si u, entonces v" - viene a ser equivalente a decir "si u es cierto, entonces v es cierto", o, lo que es lo mismo, "u es condición suficiente - - (aunque no necesaria) de v"; por tanto, el hecho de que u sea cierto y v falso viene a refutar la expresión "u→v", es decir, a hacerla falsa.

c) u = 0, v = 1. El sentido común nos dicta que un condicional de este tipo no es ni verdadero ni falso: ¿qué sentido tiene preguntarse por la verdad o falsedad de un condicional cuando el antecedente es falso?. Pero esta respuesta del - sentido común no nos satisface, porque estamos trabajando con una lógica binaria, y, establecida una interpretación (u=0,v=1 en este caso), toda sentencia deberá tener un valor, 0 ó 1, en esa interpretación. Si una sentencia no es falsa, será cierta y viceversa. Ahora bien, en el caso que nos ocupa, el condicional no es falso. Y no es falso porque, como hemos dicho antes, "u→v" es como decir que u es condición suficiente *pero no necesaria* de v, es decir, que no es la única condición, por lo que perfectamente puede ser v cierto siendo y falso. Es decir, la falsedad del antecedente no hace falso al condicional, y si no lo hace falso, lo hace cierto. (Recuérdese el ejemplo anterior: puede no haber niebla, y, sin embargo, no verse).

d) u = 0, v = 0. Pasa algo parecido al caso anterior: la condición no se da, por lo que v puede ser tan verdadero como falso, y el condicional, al no ser falso, será verdadero. - Obsérvese, además, que este empleo del condicional se encuentra en el lenguaje coloquial para señalar que, ante un dislate, - cualquier otro está justificado: "si Fulano de Tal es demócrata, yo soy el Emperador de Asiria".

Esta interpretación del condicional, aunque universalmente admitida, ha ocasionado y sigue ocasionando numerosos - problemas en Lógica y es una fuente de paradojas. Por ejemplo,

el mismo Axioma 1 del conjunto de axiomas que dimos en el Apartado 1.3 (axiomas que son tales en función de esta interpretación del condicional; para otra ya no lo podrían ser) es considerado por muchos autores como una paradoja, ya que viene a decir que si una sentencia es verdadera, todo condicional en el que esa sentencia sea el consecuente será verdadero, independientemente de que el antecedente sea verdadero o falso.

Parecidas consideraciones pueden hacerse (e invitamos al lector a reflexionar sobre ello) acerca del bicondicional (llamado a veces "equivalencia").

2.3. Tautologías y contradicciones.

Def. 2.3.1. Se dice de una sentencia que es una tautología si el resultado de su interpretación es 1 para todos los sistemas de valores de verdad (es decir, si la columna que corresponde a la sentencia en su tabla de verdad es una columna de "unos").

Def. 2.3.2. Se dice de una sentencia que es una contradicción si el resultado de su interpretación es 0 para todos los sistemas de valores de verdad (es decir, si la columna correspondiente es una columna de "ceros").

Puede comprobarse que todos los axiomas son tautologías (en un ejemplo anterior lo hemos comprobado con el axioma 3, al hallar su tabla de verdad).

Anteriormente hemos visto cómo pueden demostrarse sentencias utilizando los axiomas y las reglas de sustitución y de modus ponens y sin salirse del campo de la pura sintaxis. Tales demostraciones son bastante laboriosas, como pudo verse en el ejemplo 1.3.1. Acudiendo a la semántica, existe un teorema muy útil que vamos a enunciar, omitiendo su demostración:

Teorema 2.3.3. Toda sentencia demostrable es una tautología, y viceversa.

Gracias a este teorema puede comprobarse muy fácilmente si una sentencia es demostrable, sin necesidad de hallar su prueba formal.

2.4. Ejemplos.

2.4.1. Ejemplo 1.

Veamos cómo los razonamientos de la lógica clásica pueden formalizarse y ser analizados semánticamente a la luz de la lógica formal de predicados. Un razonamiento elemental (R) está formado por dos premisas (P1 y P2) y una conclusión (C). Para que R sea correcto, si P1 y P2 son ciertas ($P1 \wedge P2 = 1$), entonces C también deberá serlo; si alguna de ellas (o ambas) es falsa ($P1 \wedge P2 = 0$), entonces C puede ser cierta o falsa. Dicho de otro modo, lo único que no está permitido es que sea $P1 \wedge P2 = 1$ y $C = 0$: en este caso, el razonamiento no es válido ($R = 0$); en cualquier otro caso, $R = 1$. El razonamiento entonces se formaliza como un condicional:

$$(P1 \wedge P2) \rightarrow C$$

Para unos contenidos concretos de P1, P2 y C, habrá que ver si este condicional es una sentencia demostrable (es decir, bastará ver si es una tautología, en virtud del Teorema 2.3.3); si lo es, el razonamiento es correcto.

Por ejemplo (FERRATER, 1.955):

P1: 'Si Bernardo se casa, entonces Florinda se suicida'.

P2: 'Florinda se suicida si y sólo si Bernardo no se hace monje'

C: 'Si Bernardo se casa, entonces no se hace monje'.

Para formalizar este razonamiento, vamos a definir las variables proposicionales:

c = 'Bernardo se casa'

s = 'Florinda se suicida'

m = 'Bernardo se hace monje'.

Los elementos del razonamiento quedarán así:

$P1 = c \rightarrow s$

$P2 = s \leftrightarrow \neg m$

$C = c \rightarrow \neg m$

y la expresión formal del razonamiento vendrá dada por la sentencia:

$$(c \rightarrow s) \wedge (s \leftrightarrow \neg m) \rightarrow (c \rightarrow \neg m)$$

Veamos si es una sentencia demostrable; para ello, construyamos su tabla de verdad:

c	s	m	$c \rightarrow s$	$s \leftrightarrow \neg m$	$(c \rightarrow s)$ $(s \leftrightarrow \neg m)$	$c \rightarrow \neg m$	$(c \rightarrow s) \wedge (s \leftrightarrow \neg m) \rightarrow (c \rightarrow \neg m)$
0	0	0	1	0	0	1	1
0	0	1	1	1	1	1	1
0	1	0	1	1	1	1	1
0	1	1	1	0	0	1	1
1	0	0	0	0	0	1	1
1	0	1	0	1	0	0	1
1	1	0	1	1	1	1	1
1	1	1	1	0	0	0	1

Comprobamos así que es una tautología, y, por consiguiente, una sentencia demostrable, por lo que el razonamiento es co-

rrecto.

2.4.2. Ejemplo 2 (GILBERT, 1.976).

Consideremos un político que declara en la prensa:

P1: 'Si los impuestos suben, la inflacción bajará si y sólo si la peseta no se devalúa',

para,posteriormente, decir en televisión:

P2: 'Si la inflacción baja o si la peseta no se devalúa, los impuestos no subirán',

y afirmar en el Congreso:

P3: 'O bien baja la inflacción y se devalúa la peseta, o bien los impuestos deben subir'.

Nuestro político publica luego un informe en el que, tras analizar tales aseveraciones, saca de ellas la conclusión:

C: 'Los impuestos deben subir, pero la inflacción bajará y la peseta no se devaluará'.

Nos preguntamos si tal conclusión es consistente con las premisas, sin entrar en la validez de éstas. Es decir, P1, P2 y P3 pueden ser verdaderas o falsas; si alguna es falsa, la conclusión no tiene por qué ser cierta, pero si las tres son verdaderas, C debe serlo también. En definitiva, para que la inferencia de C a partir de P1, P2 y P3 sea correcta, $(P1 \wedge P2 \wedge P3) \rightarrow C$ deberá ser una tautología.

Si definimos las variables proposicionales

p = 'los impuestos suben'

q = 'la inflacción baja'

r = 'la peseta no se devalúa'.

la sentencia a comprobar será:

$$((p\rightarrow(q\leftrightarrow r))\wedge((q\vee r)\rightarrow\neg p)\wedge(q\wedge\neg r)\vee p))\rightarrow(p\wedge q\wedge r)$$

Dejamos al lector la construcción de la tabla de ver-- dad de esta sentencia. Podrá comprobar que no es una tautolo-- gía (por lo que la conclusión no es correcta), y estudiar los dos casos que hacen que el razonamiento no sea válido.

CAPITULO 3.

LOGICA DE CLASES Y ALGEBRA DE BOOLE

1. INTRODUCCIÓN.

En este Capítulo vamos a entrar en un nuevo tema: la lógica de clases. Los conceptos y la terminología que manejaremos son bastante diferentes de los que hemos visto hasta ahora. Van a ser fáciles de comprender porque la lógica de clases está muy relacionada con la teoría de conjuntos y la mayoría de esos conceptos se utilizan en álgebra. Unión, intersección, inclusión, pertenencia.... son conceptos que ya nos son familiares. En vez de utilizar conjuntos abstractos nos moveremos entre clases que son un tipo especial de conjuntos. No deberemos confundir por tanto clases y conjuntos, ni lógica de clases con teoría de conjuntos: formalmente, son iguales pero la primera es una aplicación de la segunda al campo de la lógica. Tampoco deberemos confundir una clase con un conjunto de elementos pertenecientes a una cierta clase. Un conjunto de 10 lápices no es lo mismo que la clase de los lápices, que es un concepto abstracto. Ni siquiera el conjunto de todos los lápices que existen lo sería.

2. LAS CLASES Y SUS PROPIEDADES.

La definición de clase no presenta excesivas dificultades. Para nosotros una clase va a ser un objeto abstracto que simbolizamos por las letras mayúsculas A, B, C.... . Estos objetos cumplen ciertas leyes que veremos más adelante. Estas leyes serán las de los conjuntos básicamente. Si tratásemos de investigar el contenido semántico de las clases podríamos encontrar ciertos problemas. Una clase intuitivamente la aso-

ciamos a una propiedad. La clase de las manzanas por ejemplo, la asociamos a la propiedad de crecer en los manzanos, la de los pendientes con la propiedad de colgar de las orejas..... Hay autores que identifican la clase con una propiedad. En cambio otros afirman que existen diferencias. De todas formas a nosotros esta polémica no nos va a afectar mucho. Nuestro campo se va a centrar en el aspecto sintáctico de la lógica de clases y no en el semántico.

2.1. Clases especiales.

Existen dos clases que tienen gran importancia para el desarrollo posterior. Estas son la clase de todas las clases y la clase vacia. La primera es una clase que contiene a todas las demás clases. Es decir que no hay elemento que no pertenezca a esta clase. Se representa por "U", o, a veces, por "1".

La clase vacia es una clase que representa todo lo contrario. Esta clase no engloba ningún elemento. Se representa por "0" generalmente.

2.2. Operadores y predicados.

Los operadores entre clases son tres:

"U" - unión

"∩" - intersección

"-" - complementación.

Ya los conocemos de teoría de conjuntos, por lo que no se va a hacer más hincapie en ellos. Es interesante notar el paralelismo entre la unión "U" y la conectiva "y" en la lógica proposicional o entre la intersección y la disyunción o entre la complementación y la negación. Este paralelismo se debe, co

mo veremos en el Capítulo 7, a que ambas tienen la misma estructura, una estructura de álgebra de Boole.

Los predicados(*)nos indican relaciones entre clases. Estos son:

"$\subset$" - inclusión

"$=$" - igualdad

"$\neq$" - desigualdad

Tampoco los comentaremos por ser signos habituales de la teoría de conjuntos.

Ejemplo:

En el primer capítulo hablamos del famoso ejemplo de - razonamiento que aparece en todos los libros de filosofia:

"Si Sócrates es hombre y todos los hombres son mortales entonces Sócrates es mortal".

La lógica de clases nos permite analizar mejor la estructura del razonamiento. Tenemos tres clases: S, H y M, las clases de Sócrates, los hombres y los mortales respectivamente.

La formalización según la lógica de clases de este - - enunciado sería:

$$(S \subset H) \wedge (H \subset M) \longrightarrow (S \subset M).$$

---

* La palabra "predicados" no tiene el mismo significado que - en lógica de predicados.

Como se ve hemos entrado en el contenido de las proposiciones. S $\subset$ H, H $\subset$ M y S $\subset$ M los representábamos por "x", - "y" y "z". Esto se debe a que ahora analizamos las expresiones y sentencias según otro concepto diferente. Antes simplemente nos interesaban unos enunciados y si eran falsos o verdaderos. Ahora nos interesan las clases que manejamos y sus interrela-- ciones.

También es interesante comentar por qué aparecen las - conectivas "$\wedge$" y "$\rightarrow$" en nuestra expresión de clases. Como dijimos en el primer capítulo, la lógica es el lenguaje en el - que se expresan las matemáticas. La lógica proposicional es la base de la lógica a su vez. Otros tipos de lógica son amplia-- ciones de esta primera. Por lo tanto, necesitaremos añadir a - los símbolos antes definidos los propios de la lógica proposicional para tener ya todos los elementos de nuestro lenguaje.

3. ALFABETO.

Ya hemos visto el significado de cada símbolo en el - apartado precedente. Aquí vamos a enumerarlos de una forma ordenada para tener una visión de conjunto.

a) las clases A, B, C, D,.....

   U, O, las clases universal y nula.

b) predicados

   "$\subset$" inclusión

   "$=$" igualdad

   "$\neq$" desigualdad

c) operadores

"$\cup$" - unión

"$\cap$" - intersección

"$\overline{}$" - complementación.

Para expresar propiedades y probar teoremas a veces habrá que acudir a la lógica proposicional como se ha visto en el ejemplo anterior; por ello, en realidad, el alfabeto en la lógica de clases es el que acabamos de definir, complementado con el de la lógica proposicional.

4. Diagramas de Venn.

Existe una interpretación gráfica muy sencilla de gran ayuda para entender la lógica de clases. Son los diagramas de Venn, que el lector ya conocerá por su uso en teoría de conjuntos. Aquí los vamos a aplicar para analizar razonamientos y silogismos. Hay que tener siempre presente que aquí los diagramas no representan conjuntos sino clases.

La clase de todas las clases, U, se representa por un rectángulo en cuyo seno se dibujan las clases en forma de círculos, generalmente.

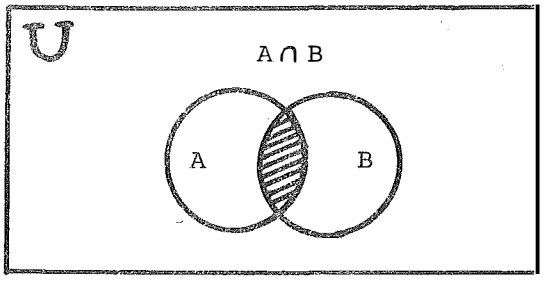


Fig. 1.

En la figura 1 la zona rayada representaria la intersección de A y B.

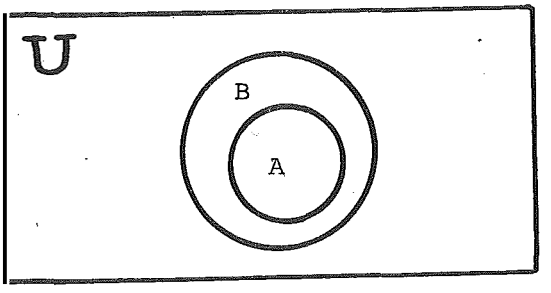
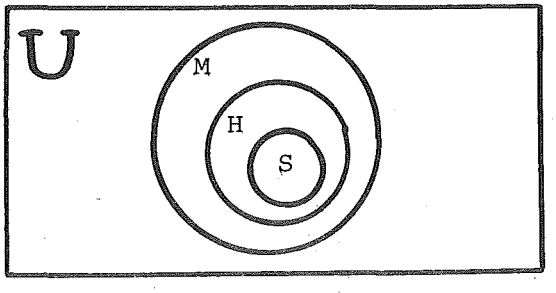


Fig. 2.

En la figura 2 se representaría que la clase A está incluida en la clase B.

Volvamos a nuestro ejemplo de Sócrates:

$$(S \subset H) \wedge (H \subset M) \rightarrow S \subset M$$



Como se ve en la figura 3, si la clase H está contenida en la M y la clase S está contenida en la H, se deduce fácilmente que la clase S está contenida en la M.

Estos diagramas se pueden pues utilizar para analizar los silogismos aristotélicos clásicos, y el lector interesado no encontrará especiales dificultades para hacerlo.

5. ALGEBRAS DE BOOLE.

Una parte de las matemáticas que tiene gran importancia en la técnica actual es el estudio de las estructuras algebráicas, que se obtienen definiendo unas operaciones sobre un conjunto. Así el conjunto se dota de una serie de entrelazamientos internos que pueden ser de gran utilidad en la representación de problemas. Los elementos de tales conjuntos son totalmente abstractos y por lo tanto generales.

Las álgebras de Boole son estructuras algebraicas de este tipo. Su creador fue George Boole, matemático inglés del siglo pasado. Las creó tratando de simular las leyes del pensamiento de una forma algebraica. Su primera aplicación aparece en lógica proposicional. Como se verá más adelante los conjuntos $\{0,1\}$ y S (conjunto de todas las sentencias), dotados de las oportunas operaciones, tienen estructura de álgebra de Boole. Pero su importancia no se queda aquí. La lógica de clases tiene estructura de álgebra de Boole, así como el conjunto de las partes de un conjunto y la estructura de los circuitos de conmutación. Existen todavía más ejemplos pero estos son los más importantes y los que vamos a ver aquí. Como se ve las álgebras de Boole tienen grandes aplicaciones, pero tampoco resuelven todo. La lógica de predicados, que mencionábamos de pa

sada en el primer capítulo, en cambio, no tiene estructura de - álgebra de Boole. Cuando estudiamos circuitos con memoria, es decir autómatas, no lo podemos hacer con estas álgebras tampoco.

Para definir qué es un álgebra de Boole vamos a volver a utilizar el método axiomático. Daremos los axiomas básicos y las propiedades y leyes que deben cumplir. Existen otras formas de hacer esta definición, en base a dotar de estructura de retículo o de anillo a un conjunto determinado, pero esta forma es la más sencilla y de la que más facilmente se desprenden sus propiedades.

Def. 5.1. Llamamos Algebra de Boole a la estructura $< B, +, ., - >$ formada por un conjunto B sobre el que definimos tres operaciones, dos binarias "+" y "." y una unitaria "-" que satisfarán las siguientes condiciones

1) $(a+b) + c = a + (b+c)$
2) $(a.b).c = a.(b.c)$ — propiedad asociativa

3) $a + b = b + a$
4) $a.b = b.a$ — propiedad conmutativa

5) $a + (b.c) = (a+b).(a+c)$
6) $a.(b + c) = a.b + a.c$ — propiedad distributiva

7) existe un elemento "0" tal que $a + 0 = a$
8) existe un elemento "1" tal que $a.1 = a$ — existencia de elementos neutros

9) $a + \bar{a} = 1$
10) $a.\bar{a} = 0$ — Existencia de un elemento complementario o inverso ($\bar{a}$ se denomina complementario o inverso de a).

De aquí es fácil demostrar las leyes más importantes de las álgebras de Boole:

Teorema 5.2. Para todo conjunto B, donde definimos una operación binaria, *, si existe elemento neutro éste es único.

Demostración: Supongamos que existen dos elementos neutros, e y e'

e = e * e' por ser e' elemento neutro

e * e' = e' por ser e elemento neutro

luego e = e'

Teorema 5.3. Dado un álgebra de Boole $\langle B, +, ., - \rangle$, para todo elemento a perteneciente a B el elemento complementario $\bar{a}$ es único.

Demostración: Supongamos que existen dos elementos b y c que cumplen a.b = 0, a+b = 1, a.c = 0 y a+c = 1

b = b+0 = b+(a.c) = (b+a).(b+c) = 1.(b+c) = b+c

c = c+0 = c+(a.b) = (c+a).(c.b) = 1.(c+b) = c+b = b+c

luego b = c

Teorema 5.4. Para todo elemento a,b perteneciente a un álgebra de Boole $\langle B, +, ., - \rangle$, se cumplen las siguientes relaciones:

1) $a.0 = 0$
2) $a + 1 = 1$
3) $a.a = a$ — Propiedad de idempotencia
4) $a + a = a$ — Propiedad de idempotencia
5) $a + a.b = a$ — Leyes de absorción
6) $a.(a + b) = a$ — Leyes de absorción
7) $\overline{a + b} = \bar{a}.\bar{b}$ — Leyes de de Morgan
8) $\overline{a.b} = \bar{a} + \bar{b}$ — Leyes de de Morgan
9) $\overline{(\bar{a})} = a$

Demostración:

1) $a.0 = a.0 + 0 = a.0 + a.\bar{a} = a.(0+\bar{a}) = a.\bar{a} = 0$

2) por dualidad

$a + 1 = (a + 1).1 = (a+1).(a+\bar{a}) = a+(1.\bar{a}) = a+\bar{a} = 1$

3) $a.a = a.a+0 = a.a+a.\bar{a} = a(a+\bar{a}) = a.1 = a$

4) $a + a = (a+a).1 = (a+a)\ (a+\bar{a}) = a+a.\bar{a} = a+0 = a$

5) $a+a.b = a.1+a.b = a(1+b) = a.1 = a$

6) $a(a+b) = a.a + a.b = a+a.b = a$

7) Si probamos que $(a+b) + \bar{a}.\bar{b} = 1$ y $(a+b).(\bar{a}.\bar{b}) = 0$ habremos demostrado que $a + b$ y $\bar{a}.\bar{b}$ son complementarios de donde se deduce facilmente que se cumple la primera ley.

$(a+b) + \bar{a}.\bar{b} = ((a+b)+\bar{a})((a+b)+\bar{b}) = (a+\bar{a})+b\ .\ a+(b+\bar{b})$

$= (1+b).(1+a) = 1$

$(a+b).(\bar{a}.\bar{b}) = a.(\bar{a}.\bar{b}) + b.(\bar{a}.\bar{b}) = (\bar{a}.a)\ \bar{b}+\bar{a}\ (b.\bar{b}) =$

$= 0.\bar{b} + 0.\bar{a} = 0$

8) Siguiendo un camino paralelo se demuestra facilmente.

9) $\overline{(\bar{a})}$ es el complemento de $\bar{a}$ y como a también es complemento de $\bar{a}$ y éste debe de ser único.

$$a = \overline{(\bar{a})}$$

Esta forma de definir un álgebra de Boole tiene la ventaja de resaltar las llamadas formas duales.

Vemos que las leyes y teoremas se dan siempre por pares y que uno es el resultado de sustituir la suma por el pro

ducto y el producto por la suma en el otro.

Esta propiedad se podría haber utilizado en la demostración de teoremas, dado que, probado uno, el dual también lo está.

6. ESTRUCTURA ALGEBRAICA DE LA LÓGICA DE CLASES.

La lógica de clases vimos que está compuesta por un conjunto universal U en el que se incluyen toda una serie de clases. Entre estas clases hemos definido unas operaciones que denominamos intersección, unión y complementación. También postulamos la existencia de la clase vacía. La unión, intersección y complementación de clases, formalmente son idénticas a las de los conjuntos. Todos sabemos que estas operaciones cumplen las siguientes propiedades.

$$(A \cup B) \cup C = A \cup (B \cup C)$$

$$(A \cap B) \cap C = A \cap (B \cap C)$$

$$A \cup B = B \cup A$$

$$A \cap B = B \cap A$$

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

$$A \cup \bar{A} = U$$

$$A \cap \bar{A} = 0$$

$$0 \cup A = A$$

$$U \cap A = A$$

De aqui se puede deducir que el conjunto compuesto por la clase universal, la clase nula y todas las demás clases existentes, con la unión, intersección y complementación definidas en su seno tiene estructura de álgebra de Boole. Si denominamos C al conjunto anterior, este álgebra de Boole se representaría por $\langle C, \cup, \cap, - \rangle$. Es un álgebra que puede ser finita o infinita según la cantidad de clases que incluyamos en C .

Como corolario se puede deducir que el conjunto de las partes de un conjunto tiene estructura de álgebra de Boole también.

7. Algunos ejemplos de álgebras de Boole.

Existen gran cantidad de ejemplos prácticos de aplicaciones de álgebras de Boole. Tres de ellos tienen gran importancia para nosotros. El primero, el álgebra de Boole binaria, es de gran utilidad en electrónica digital. También se suele denominar álgebra de conmutación.

Está compuesto por dos elementos y, como en todo álgebra de Boole debe existir elemento nulo y universal, uno de ellos será el primero y otro el segundo. Es decir que el álgebra estará compuesta por los elementos 0 y 1. Se puede comprobar fácilmente que estos dos elementos son complementarios. Las operaciones suma y producto son fáciles de analizar, deben regirse por las propiedades de los elementos de un álgebra de Boole.

Vamos a ver cómo se opera con los elementos del álgebra $\langle \{0,1\} , +, ., - \rangle$

a) Complementación.

$$\bar{1} = 0$$

$$\bar{0} = 1$$

b) Suma booleana, a + b

$0 + 0 = 0$ (Idempotencia)

$0 + 1 = 1$

Por complementarios

$1 + 0 = 1$

$1 + 1 = 1$ (Idempotencia)

c) Producto booleano a.b

$0.0 = 0$ (Idempotencia)

$0.1 = 0$

Por complementarios

$1.0 = 0$

$1.1 = 1$ (Idempotencia)

Se aprecia enseguida un paralelismo entre estas operaciones y la interpretación en forma de tablas de verdad de las conectivas "$\neg$", "$\vee$" y "$\wedge$", respectivamente.

Otro caso muy importante de conjunto con estructura de este tipo es el conjunto de las partes de un conjunto, con las operaciones de complementación, reunión e intersección, como hemos visto en el apartado anterior.

Otro ejemplo que posteriormente nos va a ser de gran utilidad es el formado por el siguiente conjunto: Dentro del conjunto de todas las sentencias, definimos una relación de

equivalencia, R, entre dos sentencias A y B, de forma que A R B si A y B tienen la misma interpretación en {0,1} para cualquier interpretación posible de las variables. Como se ve hemos realizado una partición en clases de equivalencia dentro de este conjunto. Esta partición tiene estructura de álgebra de Boole para las operaciones $\wedge$, $\vee$ y $\neg$. Las operaciones $\longrightarrow$, $\longleftrightarrow$ se pueden definir en función de las tres primeras como se verá en capítulos posteriores.

CAPITULO 4.

FORMAS BOOLEANAS

1. INTRODUCCIÓN.

Las formas booleanas son entes matemáticos abstractos - de gran interes. Su definición se va a hacer de forma totalmente general para luego pasar a ver sus aplicaciones. Las relaciones entre formas booleanas y funciones booleanas, de gran importancia en el álgebra de conmutación, se verán en el siguiente - capítulo.

Un paralelismo que se apreciará a primera vista es el existente entre las sentencias y las formas booleanas. Esto se debe a la estructura de forma booleana que tienen las senten-- cias. El conjunto de variables proposicionales y sentencias se verá más adelante que es isomorfo con el de variables booleanas y formas booleanas. Las dos conectivas "$\rightarrow$" y "$\leftrightarrow$" se pueden definir en función de "$\wedge$", "$\vee$" y "$\neg$". Llegándose así a la conclusión de que las sentencias son una aplicación particular de - - unas estructuras más generales llamadas formas booleanas. Esto se estudiará en el capítulo 7.

Al principio del capítulo se estudia la relación de orden que se puede introducir dentro de un álgebra de Boole y la estructura de retículo que le confiere esta relación, relación no muy importante en álgebra de conmutación o en lógica proposicional pero sí en otros campos. En teoria de conjuntos, por ejemplo, esta relación se llama inclusión y todos conocemos su importancia.

2. Relaciones de orden dentro de las álgebras de Boole.

Una relación de orden parcial es una relación binaria entre los elementos de un conjunto B dotada de las propiedades reflexiva, antisimétrica y transitiva, es decir:

$$a \; R \; a, \forall \; a \in B$$

$$(a \; R \; b) \wedge (b \; R \; a) \longrightarrow (a = b), \forall \; a, \; b \in B$$

$$(a \; R \; b) \wedge (b \; R \; c) \longrightarrow (a \; R \; c), \forall \; a, \; b, \; c \in B$$

En un álgebra de Boole se puede establecer la siguiente relación de orden:

Def. 2.1. Dado el álgebra de Boole $\langle B, +, ., - \rangle$, decimos

$$a \leq b \text{ si y sólo si } a.b = a, \forall \; a, \; b \in B$$

Teorema 2.2. Dado el álgebra de Boole $\langle B, +, ., - \rangle$ tal que $a, b, c \in B$, se verifica que:

1°) $a \leq a$

2°) $(a \leq b) \wedge (b \leq a) \longrightarrow (a = b)$

3°) $(a \leq b) \wedge (b \leq c) \longrightarrow (a \leq c)$

(Por consiguiente, $\leq$ es una relación de orden en B)

Demostración

1°) $a.a = a$

2°) $(a \leq b) \longrightarrow (a = a.b)$

$(b \leq a) \longrightarrow (b = b.a = a.b = a)$

3°) $(a \leq b) \longrightarrow (a.b = a)$

$(b \leq c) \longrightarrow (b.c = b)$; $a.(b.c)=(a.b).c=a.c=a.b=a$

Teorema 2.3. Dado un álgebra de Boole $\langle B, +, ., - \rangle$ tal que $a,b \in B$, $a.b = a$ es equivalente a: $a + b = b$

Demostración:

Si $a.b = a$, entonces $a+b = a.b+b = b$ (por absorción)

Teorema 2.4. Dados los elementos a, b, c pertenecientes a un álgebra de Boole $\langle B, +, ., - \rangle$, las siguientes relaciones son válidas:

1°) $a.b \leq a$

2°) $a \leq a + b$

3°) $(a \leq c) \wedge (b \leq c) \longrightarrow a + b \leq c$

4°) $a \leq b$ si y sólo si $a.\bar{b} = 0$

5°) $0 \leq a$ y $a \leq 1$ para todo a

Demostración:

1°) $(a.b).a = (a.a).b = a.b$ es decir $a.b \leq a$

2°) $a.(a + b) = a$ es decir $a \leq a + b$

3°) $(a + b).c = a.c + b.c = a + b$

4°) Si $a \leq b$ se cumple $a.b = a$ y $a.\bar{b} = a.b.\bar{b} = a.0=0$

Si $a.\bar{b} = 0$ entonces $a \leq b$ porque

$a = a.1 = a.(b + \bar{b}) = a.b + a.\bar{b} = a.b$

5°) $0.a = 0$ y $a.1 = a$

Dentro del álgebra hemos visto que existe una relación de orden; además podemos establecer la existencia de un extremo inferior y un extremo superior para cada par de elementos - a, b, de la siguiente forma:

Extremo sup. de (a,b) = a + b

Extremo inf. de (a,b) = a . b

de donde se deduce que un álgebra de Boole tiene estructura de retículo.

3. VARIABLES BOOLEANAS.

Las variables booleanas representan a los elementos - del álgebra de Boole sobre la que están definidas, y se designan con las letras x, y, z, a veces con subíndices. En el conjunto A de variables booleanas se definen las mismas operaciones que en el conjunto B de elementos a los que representan.

Def. 3.1. Llamamos valor de una variable al elemento que representa en un caso determinado.

Obsérvese que hemos seguido un proceso inverso al de la lógica proposicional: allí, primero definimos U (conjunto de todas las sentencias) y luego la interpretación <I,V>, donde V era una conjunto arbitrario e I una aplicación I:U→V; - aquí, primero se ha definido B (equivalente a V, el conjunto de interpretación) y luego A (equivalente a U). Aquí también tendremos una aplicación α:A→B, equivalente a I, que más adelante definiremos como "función de asignación".

4. FORMAS BOOLEANAS.

Siguiendo un proceso paralelo al que utilizamos en lógica proposicional, donde partimos de las variables proposicionales para llegar a las sentencias, vamos a partir aqui de las variables booleanas para llegar a las formas booleanas. Aquí en vez de conectivas tenemos operaciones entre variables. Además tenemos 3 en vez de 5. Hay tres conectivas, "$\wedge$", "$\vee$" y "$\neg$" que intuitivamente tienen gran similitud a ".", "+" y "-" respectivamente. Sus tablas de verdad se vio en el apartado 6 del capítulo 3 que coinciden con el resultado de estas operaciones en un álgebra de Boole binaria. Nos quedan el condicional y el bicondicional. Como veremos mas adelante (Cap. 7), mediante las tres operaciones anteriores podemos definir otras dos que sean las equivalentes al condicional y al bicondicional.

En realidad un álgebra de Boole se puede definir con dos operaciones únicamente, la suma y la complementación, o producto y la complementación, y definir las demás en función de estas dos.

Este método es menos intuitivo que el seguido por nosotros y por eso no lo hemos utilizado. En lógica proposicional también se pueden establecer solamente dos conectivas, "$\wedge$" y "$\neg$" o "$\vee$" y "$\neg$" y definir las tres restantes en función de cualesquiera de estas dos. Más adelante se verá la forma de hacerlo.

4.1. Alfabeto.

Vamos a enumerar los signos que utilizaremos para construir las formas booleanas.

a) variables booleanas: "x", "y", "z".....

b) operaciones: "+", "." y "-"

c) signos de puntuación: "(",")" (paréntesis)

4.2. Expresiones y Formas booleanas.

Def. 4.2.1. Llamamos expresión booleana a toda secuencia finita de símbolos de nuestro alfabeto, colocados uno al lado de otro.

Def. 4.2.2. Llamamos forma booleana a toda expresión A_i que cumple una de las tres condiciones siguientes

1°) A_i es una variable booleana.

2°) A_i es el 0 ó el 1 del álgebra de Boole

3°) Si A_j y A_k son formas booleanas, A_i es la forma $\overline{A}_j$, ó $A_j.A_k$, ó $A_j + A_k$.

4.3. Funciones de asignación y valuación.

Vamos a establecer una forma de valuación de las formas que nos permita hallar el valor que toman éstas en función del que toman las variables.

Def. 4.3.1. Llamamos función de asignación α a toda aplicación del conjunto de variables en el conjunto B de elementos del álgebra de Boole $\langle B, +, ., - \rangle$ sobre el que toman valor las variables.

Esta función de asignación es el equivalente a dar valores a las variables. El valor que toma una variable según α se denomina por $\alpha\ (x_i)$. Las funciones de asignación de formas construidas a partir de n variables se pueden representar tambien por n-tuplos de B^n siendo B^n el producto cartesiano de orden n de B consigo mismo.

Def. 4.3.2. Dada una forma booleana, llamamos función de valuación relativa a la función de asignación α a la aplicación de A_i en B. El elemento final de B se establece recursivamente según las siguientes reglas (la función de valuación de A_i con respecto a α se representa por $|A_i|\alpha$):

1. $|0|\alpha = 0$, $|1|\alpha = 1$

2. Si x_i es una variable $|x_i|\alpha = \alpha(x_i)$ para $i = 1,2,..n$

3. Si A_i y A_j son formas booleanas entonces $|\bar{A}_i|_\alpha = \overline{|A_i|_\alpha}$

$$|A_i.A_j|_\alpha = |A_i|_\alpha .|A_j|_\alpha \quad y \quad |A_i+A_j|_\alpha = |A_i|_\alpha + |A_j|_\alpha$$

Obsérvese la similitud que tiene la asignación de formas booleanas con la aplicación I que definimos cuando establecimos qué era la interpretación de una sentencia en lógica proposicional. En realidad es lo mismo, como ya se verá mas adelante cuando estudiemos la relación que existe entre álgebra de Boole y lógica proposicional (Cap. 7).

Ejemplo: Tenemos el álgebra de Boole binaria $\langle\{0,1\}, +, ., -\rangle$ y las variables x, y, z que pueden tomar los valores "0" y "1"

$(x + y).z$; $(x.y) + (y.z)$ son formas booleanas.

Si definimos α como $x = 1$, $y = 0$, $z = 0$

La función de valuación dará:

$$(1 + 0). 0 = 0$$

$$(1.0) + (0.0) = 0$$

Las variables representan elementos de un álgebra de

Boole. Esto implica que deben cumplir las mismas propiedades que se establecen en la definición para los elementos. En - - nuestro caso tales propiedades establecen una equivalencia entre las diferentes formas booleanas que se representa por el signo "=". Así:

$A_i.A_i = A_i$ ó $A_i + A_j = A_j + A_i$,.... siendo A_i y A_j formas booleanas.

El resultado de esta relación de equivalencia definida en el conjunto de las formas booleanas es una partición en clases de equivalencia. A_i y A_i. A_i por ejemplo pertenecerán a la misma clase de equivalencia. Esta relación confiere una estructura al conjunto de las formas booleanas, que pasamos a estu-- diar a continuación.

Llamamos B'_n al conjunto de todas las formas construidas a partir de n variables, y $B_n = B'_n/=$ a la partición establecida en B'_n por la relación de equivalencia

Teorema 4.3.3. El sistema $\langle B_n, +, ., - \rangle$ formado por el conjunto B_n de todas las clases de equivalencia y las operaciones "+", "." y "-" definidas como operaciones entre clases tiene estructura de álgebra de Boole.

Demostración:

Entre las formas booleanas están definidas las opera-- ciones; por lo tanto entre las clases también:

$$\left.\begin{array}{l} A, E \in \text{clase } 1 \longrightarrow A = E \\ B, D \in \text{clase } 2 \longrightarrow B = D \end{array}\right\} \longrightarrow A + B = E + D$$

De aquí se deduce que podemos operar directamente con clases:

clase 1 + clase 2 = clase 3.

Las operaciones entre clases nos dan otras clases, luego son cerradas.

Todas las formas booleanas incluidas en una clase, para cualquier función de asignación representan el mismo elemento, aunque éste varíe de una función α a otra para todas a la vez. Así las clases deberán tener las propiedades que tienen los elementos de un álgebra de Boole. (Esto se puede comprobar propiedad a propiedad).

Luego $\langle B_n, +, ., - \rangle$ es un álgebra de Boole.

Según la definición dada de función de valuación, dos formas equivalentes tendrán la misma función de valuación para la misma α.

4.4. Formas canónicas.

Hemos visto en la sección precedente las formas booleanas y sus clases. Podemos determinar si dos elementos pertenecen a una misma clase pero no tenemos una caracterización precisa de cada clase. El problema se aborda del siguiente modo:

4.4.1. Conceptos previos

Def. 4.4.1.1. Llamamos elemento atómico de un álgebra de Boole $\langle B, +, ., - \rangle$ a todo elemento b, distinto de 0, perteneciente a B, que cumple la siguiente condición:

$b.a = 0$ ó $b.a = b$ para todo $a \in B$

Teorema 4.4.1.2. Dado un álgebra de Boole $\langle B, +, ., - \rangle$ cuyos elementos atómicos son $b_1 ..... b_n$, existe una expresión de la forma:

$a = b_\alpha + b_\beta + ... + b_\gamma$, tal que $1 \leq \alpha, \beta, ..., \gamma \leq n$

para cada elemento a $\neq 0$ de B, y esta expresión es única para cada elemento.

Demostración:

Sean b_α, b_β,...b_γ todos los elementos atómicos de B - menores que a. Según las propiedades expresadas en el Teorema 2.4 sobre la relación de orden tenemos $b = b_\alpha + b_\beta + \ldots b_\gamma \leq a$

Vamos a demostrar que $a.\bar{b} = 0$ y por tanto que $a \leq b$ - que junto con $b \leq a$ nos da $a = b$. Por el teorema de de Morgan,

$$a.\bar{b} = a.\bar{b}_\alpha \ldots\ldots \bar{b}_\gamma$$

Si b_ω es un elemento atómico que pertenece al conjunto $\{b_\alpha, b_\beta \ldots\ldots b_\gamma\}$ se deduce que $a.\bar{b}.b_\omega = 0$. Si b no pertenece a este conjunto entonces $a.\bar{b}.b_\omega = 0$ porque $a.b_\omega = 0$. Por tanto, deberá ser $a.\bar{b} = 0$, con lo que hemos demostrado que $a_\alpha = b_\beta + b_\gamma + \ldots + b$. Ahora tenemos que demostrar la unicidad.

Supongamos que existen $\{b_\alpha, \ldots, b_\gamma\}$ y $\{b_a, \ldots b_z\}$ tales que:

$$a = b_\alpha + \ldots\ldots + b_\gamma = b_a + \ldots + b_z$$

$b_\alpha \leq a$ por ejemplo; como en el conjunto $\{b_a \ldots\ldots b_z\}$ están todos los elementos atómicos menores que a, alguno de éstos será igual a b_α . Es decir que los dos conjuntos tendrán los mismos elementos, aunque el orden pueda ser diferente.

Hemos definido unos elementos, los elementos atómicos que son algo asi como los elementos básicos de un álgebra de - Boole. En función de éstos se pueden representar todos los demás. Si tenemos n elementos atómicos, por inducción se ve fá--

cilmente que el número de elementos de este álgebra de Boole - es 2^n. El álgebra de las clases de equivalencia de las formas booleanas también tendrá elementos de este tipo. Estos son los productos canónicos que definimos a continuación.

4.4.2. La forma canónica.

Veamos cómo se interpreta el teorema anterior en el álgebra de Boole $\langle B_n, +, ., - \rangle$. Deberán existir unas clases - que estén compuestas por elementos equivalentes que sean ele-- mentos atómicos del álgebra y en función de estos elementos - atómicos podremos expresar cualquier otra clase. El problema - está en encontrar alguna forma booleana que sea representativa de cada una de estas clases.

Supongamos que tenemos un álgebra de formas booleanas generada por n variables $x_1, x_2 \ldots\ldots x_n$. Por l_i entendemos un - nuevo tipo de variable que puede tomar el valor x_i o $\bar{x}_i$.

Def. 4.2.2.1. Dadas las variables $x_1, \ldots . x_n$ llamamos - producto canónico P_n de dichas variables a toda forma booleana construida de la siguiente forma (Π representa aquí el producto booleano):

$$P_n = l_1 . l_2 \ldots\ldots l_n = \prod_1^n l_i$$

Def. 4.2.2.2. Llamamos forma canónica a cualquier forma booleana compuesta por sumas de productos canónicos diferentes entre sí.

Teorema 4.2.2.3. Dada el álgebra de Boole $\langle B_n, +, ., - \rangle$ sus productos canónicos son elementos atómicos.

Demostración: Vamos a demostrar por inducción que cual

quier producto canónico cumple $P_n.a = P_n$ ó $P_n.a = 0 \quad \forall a \in B_n$.

Para n = 1 tenemos que existen dosproductos canónicos x, $\bar{x}$. El conjunto de formas booleanas generadas por una variable son:

0, x, $\bar{x}$, 1. El resto son equivalentes a alguna de és-- tas. Se puede comprobar fácilmente analizando caso por caso las condiciones que debe cumplir una forma booleana para serlo y - aplicando las leyes de equivalencia entre formas.

Supongamos que para n - 1 se cumple que

$$\prod_{1}^{n-1} y_i \cdot a = \prod_{1}^{n-1} y_i \quad \text{ó} \quad \prod_{1}^{n-1} y_i.a = 0 \quad \forall a \in B_n$$

Para n podemos suponer que construimos primero todas - las formas y sus clases con n-1 variables. Estas serán las mis mas que para n-1, luego cumplirán las condiciones anteriores.

Según la definición de forma booleana podemos generar nuevas formas siendo éstas 0,1, una variable, u operando for-- mas ya conocidas. Serán diferentes en los tres casos siguien-- tes:

(a_{n-1} representa una forma generada por n-1 variables).

a) y_n

b) $a_{n-1} \cdot y_n$

c) $a_{n-1} + y_n$

a) $\left(\prod_{1}^{n-1} y_i\right) \cdot y_n \cdot y'_n = \begin{cases} \prod_{1}^{n-1} y_i \cdot y_n \\ 0 \end{cases}$ según y_n e y'_n sean iguales o no

b) $\left(\prod_{1}^{n-1} y_i\right) \cdot y_n \cdot a_{n-1} \cdot y'_n = \left(\prod_{1}^{n-1} y_i \; a_{n-1}\right) \cdot \left(y_n \cdot y'_n\right) = \begin{cases} \prod_{1}^{n-1} y_i \cdot y_n \\ 0 \end{cases}$

hay cuatro posibilidades: $y_n \cdot y'_n$ puede tomar el valor y_n ó 0 y $\prod_{1}^{n-1} y_i \cdot a_{n-1}$ el valor $\prod_{1}^{n-1} y_i$ ó 0; combinándolos se obtiene el resultado anterior.

c) $\prod_{1}^{n-1} y_i \cdot y_n \; (a_{n-1} + y'_n) = \prod_{1}^{n-1} y_i \cdot a_n \cdot y_n + \prod_{1}^{n-1} y_i \; y_n \cdot y'_n = \begin{cases} \prod_{1}^{n-1} y_i \cdot y_n \\ 0 \end{cases}$

Volvemos a tener cuatro posibilidades: que el primer término tome el valor 0 ó $\prod_{1}^{n-1} y_i \; y_n$ y que el segundo tome 0 ó $\prod_{1}^{n-1} y_i \cdot y_n$. Su suma da siempre 0 ó $\prod_{1}^{n-1} y_i \cdot y_n$

El producto de dos productos canónicos siempre da 0. Si son diferentes deberán diferir en al menos un elemento. Este será la negación de alguna y_i del anterior luego su intersección da 0.

Teorema 4.2.2.4. La suma de todos los productos canónicos es 1.

Demostración:

Para n = 1 se cumple $x + \bar{x} = 1$

Para n - 1 suponemos que también.

Si P_{n-1} es un producto canónico para n-1 variables, para n todos los productos canónicos serán de la forma:

$$P_{n-1} \cdot x \quad \text{ó} \quad P_{n-1} \cdot \bar{x}$$

La suma de todos los P_n la podemos agrupar por pares:

$$P_{n-1} \cdot x + P_{n-1} \cdot \bar{x}$$

Por distributividad $P_{n-1} (x + \bar{x}) = P_{n-1}$ volvemos a obtener la suma para n-1 que toma el valor 1.

Como corolario de esta demostración se deduce que cualquier elemento atómico de B_n es equivalente a un producto canónico.

Las definiciones y teoremas anteriores nos llevan a la siguiente conclusión: cada clase de elementos atómicos de B_n se puede representar por un producto canónico y como cada elemento de un álgebra de Boole se puede expresar como suma de elementos atómicos, *podemos representar cada clase por una forma canónica*. Es decir, las funciones de valuación de una forma canónica y de cualquier forma equivalente a ella serán las mismas para cualquier función de asignación.

Un álgebra de formas generadas por n variables tiene 2^{2^n} clases diferentes: Tenemos 2^n elementos atómicos diferentes. K elementos atómicos se comprueba fácilmente por inducción que generan 2^k elementos, luego tenemos 2^{2^n} clases

CAPITULO 5.

EL ALGEBRA DE CONMUTACION

1. INTRODUCCIÓN.

Ya introdujimos en el capítulo tercero el álgebra de conmutación, como un álgebra de Boole compuesta por dos elementos únicamente. También estudiamos la forma en que se deben operar estos dos elementos entre sí. Este capítulo va a tratar de funciones dentro de álgebras de este tipo. Este tipo de funciones son de gran importancia en el estudio de circuitos lógicos. Son el soporte teórico de éstos. Un circuito lógico no es más que la simulación de una función de conmutación. Aunque no se haga mención de ello, todo lo que hablemos a continuación será una particularización al álgebra binaria.

2. FUNCIONES DE CONMUTACIÓN.

Def. 2.1. Dado un álgebra $\langle \{0,1\} , +, ., - \rangle$, llamamos función de conmutación de orden n a toda aplicación $\{0,1\}^n \longrightarrow \{0,1\}$, donde $\{0,1\}^n$ es el producto cartesiano de orden n de $\{0,1\}$ consigo mismo.

La estructura de estas funciones es como la de una función ordinaria. El conjunto inicial es el de todas las n-tuplas posibles formadas con los elementos "0" y "1"; el conjunto final está compuesto por los elementos "0" y "1" solamente. Esta forma de definir una función de conmutación permite definir operaciones entre funciones.

Def. 2.2. Dadas las funciones de conmutación de orden n, f y g, las operaciones f + g, f.g y $\bar{f}$ se definen así:

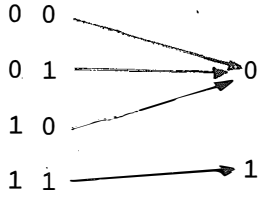
f + g y f.g son las aplicaciones resultantes de sumar o multiplicar para cada par de elementos iniciales iguales de f y g - los elementos finales de ambos. $\bar{f}$ es la aplicación resultante de hacer corresponder a cada elemento inicial de f la negación del que le corresponde según f.

Ejemplo 1.1.2.

Dos funciones de conmutación de 2° orden serían

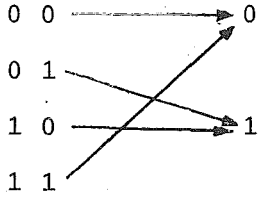
1ª función (f)

Conjunto de n-tuplas

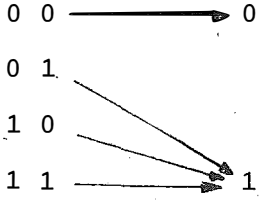


2ª función (g)

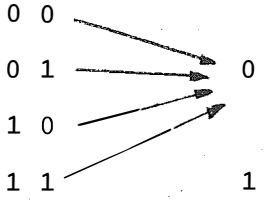
Conjunto de -n-tuplas



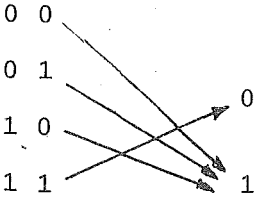
f + g



f . g



$\bar{f}$



Teorema 1.1.3. Sea F_n el conjunto de todas las funciones de conmutación de orden n. El sistema $\langle F_n, +, ., - \rangle$ tiene estructura de álgebra de Boole.

Demostración:

Las operaciones entre funciones se reducen a operaciones en el álgebra de Boole $\langle \{0,1\}, +, ., - \rangle$, luego éstas cumplen las propiedades establecidas en la definición 5.1. del capítulo 3. Al operar funciones de orden n obtenemos otras de orden n también, luego las operaciones son cerradas. Los elementos 0 y 1 son las aplicaciones de todos en 0 y de todos en 1 - respectivamente.

El número de elementos de F_n es 2^{2^n} : Tenemos 2^n n-tuplas que podemos aplicar en "0" o en "1", luego el total de - funciones de conmutación de orden n diferentes es 2^{2^n}.

Los elementos atómicos de esta aplicación se ve fácilmente cuáles son: La aplicación que hace corresponder todas - las n-tuplas a 0, excepto uno de ellos, que se corresponde con 1. En el ejemplo anterior, f era un elemento atómico del álgebra de funciones de conmutación de orden 2. Intuitivamente se ve facilmente que con sumas de aplicaciones de este tipo se - pueden obtener todas las demás.

3. REPRESENTACIÓN DE FUNCIONES DE CONMUTACIÓN MEDIANTE FORMAS BOOLEANAS.

Vamos a dar algunas definiciones previas que son necesarias para establecer el tipo de relación existente entre formas y funciones.

Def. 3.1. Decimos que dos álgebras de Boole $\langle B', +, ., - \rangle$

$\langle B", +, ., - \rangle$ son isomorfas si entre ellas existe una aplicación biyectiva $I:B' \to B"$ que cumple las tres condiciones siguientes para $a, b \in B'$

$$I\ (a + b) = I\ (a) + I\ (b)$$

$$I\ (a \cdot b) = I\ (a \cdot b)$$

$$I\ (\bar{a}) = \overline{I\ (a)}$$

La mayor semejanza que puede existir entre dos estructuras algebráicas es el isomorfismo. Por ser biyectiva ambas deben tener el mismo número de elementos y además las tres condiciones que se deben cumplir equivalen a establecer que se deben conservar todas las propiedades de la estructura algebraica del primer conjunto en el segundo.

Teorema 3.2. Dos álgebras de Boole $\langle B', +, ., - \rangle$ y $\langle B", +, ., - \rangle$ con el mismo número de elementos tienen estructuras isomorfas.

Demostración:

Si tienen el mismo número N de elementos ambas tendrán un número n de elementos atómicos tal que $N = 2^n$. Si $\gamma_1, \gamma_2 \ldots \gamma_n$ son los elementos atómicos de B' y $\beta_1, \beta_2, \ldots \beta_n$ los de B" podemos establecer una correspondencia biyectiva en ambos conjuntos de elementos atómicos y extenderla al resto de los conjuntos de la siguiente forma.

$$f\ (\gamma_j + \gamma_i + \ldots + \gamma_k) = f\ (\gamma_j) + f\ (\gamma_i) f \ldots f(\gamma_k)$$

donde f es la primitiva aplicación establecida entre elementos atómicos. Tal aplicación por definición es isomorfa.

Como los conjuntos F_n y B_n tienen el mismo número de elementos, deberán tener estructuras isomorfas. Deberá existir por tanto una correspondencia biunívoca que relacione cada clase de B_n con cada función de F_n.

En base a la función de valuación de las formas booleanas vamos a definir la aplicación β siguiente:

$$\beta = \bigcup_{\alpha} (\alpha, |A_i|\alpha) \quad \forall \alpha \in \{0,1\}^n$$

Esta aplicación así definida es una función de conmutación; los elementos iniciales son n-tuplos de $\{0,1\}^n$ y los finales, elementos de $\{0,1\}$. La función definida en base a los productos canónicos de B_n se corresponde con los elementos atómicos de F_n. Se puede probar fácilmente por inducción. El resto de las clases de B_n se pueden representar como suma de estos productos atómicos por lo que sus respectivas β serán suma de los elementos atómicos correspondientes a los productos. Así vemos que hemos establecido una aplicación isomorfa entre B_n y F_n que nos hace corresponder a cada $A_i \in B_n$ un $f_i \in F_n$ en base a la función de valuación de A_i.

Este resultado es de importancia vital en el diseño lógico porque nos permite un método sistemático de diseño de circuitos lógicos. Ahora sabemos que una forma booleana es equivalente a una función determinada. Así podemos utilizar formas que son mucho más sencillas de utilizar. La suma de dos formas, por ejemplo, representará a la función suma de las dos funciones que representaban las dos formas primitivas. Pero todavia hay un problema no resuelto: dentro de una clase de formas equivalentes, además de hallar la forma canónica, que ya conocemos, hallar la forma más sencilla equivalente a esta forma canónica. Esto tiene importancia en diseño porque cuanto más sencilla sea la forma, más sencilla y barata será su realización. Este problema se aborda en el punto 5 que habla sobre minimización.

4. Representación de funciones de conmutación mediante tablas de valores.

4.1. Tablas de valores o tablas de verdad.

Existe una forma de representar una función que es de gran utilidad en diseño: las tablas de valores, también llamadas tablas de verdad.

Una función está compuesta por un conjunto de pares ordenados, y se puede representar de dos maneras: expresando una propiedad que deban cumplir estos pares y solamente éstos, que en nuestro caso sería mediante formas booleanas. O dar todo el conjunto de pares ordenados. Como en nuestro caso éste es finito, podemos hacerlo. Este conjunto se denomina tabla de valores. La función del ejemplo 1.1.2 se representaría de la siguiente forma:

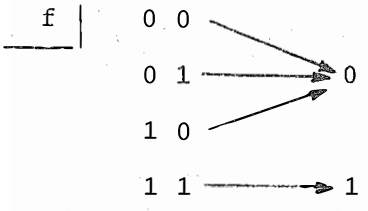


Tabla de verdad

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

como tenemos dos variables, existen cuatro n-tuplas para las cuales damos el elemento final de la aplicación.

4.2. Representación como forma booleana de una función dada por su tabla de verdad.

El problema que se plantea aquí es el siguiente: dada una tabla de verdad, cómo pasar a representar la función a través de una forma booleana.

El problema inverso es fácil: si tenemos , por ejemplo, la forma canónica $f_1 = x_1 + x_2$ asignando a f todas las n-tuplas de x_1, x_2 y obteniendo valores de la función de valuación obtenemos la tabla de valores:

x_1	x_2	$x_1 + x_2$	f_1
0	0	0 + 0	0
0	1	0 + 1	1
1	0	1 + 0	1
1	1	1 + 1	1

Para realizar la operación inversa, en cambio, deberemos hacer lo siguiente: sabemos que todas las funciones son expresables como suma de los elementos atómicos de F_n, que son aplicaciones donde todos los elementos se corresponden con 0 excepto un elemento que se corresponde con 1. La función será la suma de todos los elementos atómicos correspondientes a las n-tuplas que estén aplicadas en 1. Así la forma canónica de la función será la suma de los productos canónicos correspondientes a estas n-tuplas.

Ejemplo 4.2.1.

La función f_2 cuya tabla de verdad es:

x_1	x_2	f_2
0	0	0
0	1	1
1	0	0
1	1	1

vemos que tiene dos n-tuplas que se corresponden con 1: (0,1) y (1,1); según la función de valuación, los productos canónicos que le corresponden son $\bar{x}_1 \cdot x_2$ y $x_1 \cdot x_2$ por lo que $f_2 = \bar{x}_1 \cdot x_2 + x_1 \cdot x_2$. Los elementos atómicos de F_2 que representarían estos productos canónicos son:

x_1	x_2	$\bar{x}_1 \cdot x_2$		x_1	x_2	$x_1 \cdot x_2$		x_1	x_2	f_2
0	0	0		0	0	0		0	0	0
0	1	1	+	0	1	0	=	0	1	1
1	0	0		1	0	0		1	0	0
1	1	0		1	1	1		1	1	1

Se comprueba fácilmente que la suma de estas dos funciones da f_2.

Ejemplo 4.2.2.

Consideremos una función en F_4 cuya tabla de verdad es:

x_1	x_2	x_3	x_4	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Solamente hay tres n-tuplas que tomen el valor 1 (0, 0,1,1), (1,0,0,0) y (1,1,0,0); les corresponden los productos canónicos $\bar{x}_1.\bar{x}_2.x_3.x_4$, $x_1.\bar{x}_2.\bar{x}_3.\bar{x}_4$ y $x_1.x_2.\bar{x}_3.\bar{x}_4$, luego la -

forma canónica de f será:

$$f = \bar{x}_1 . \bar{x}_2 . x_3 . x_4 + x_1 . \bar{x}_2 . \bar{x}_3 . \bar{x}_4 + x_1 . x_2 . \bar{x}_3 . \bar{x}_4$$

5. MINIMIZACIÓN.

Cuando tenemos una función de conmutación, sea como -- forma booleana o como tabla de valores, para realizarla elec-- trónicamente nos interesa obtener de ella la forma booleana - más sencilla posible. En el ejemplo anterior vimos la función representada en la tabla adjunta, cuya forma canónica es:

$$f = \bar{x}_1 . x_2 + x_1 . x_2$$

x_1	x_2	f
0	0	0
0	1	1
1	0	0
1	1	1

Aplicando la propiedad distributiva vemos que:

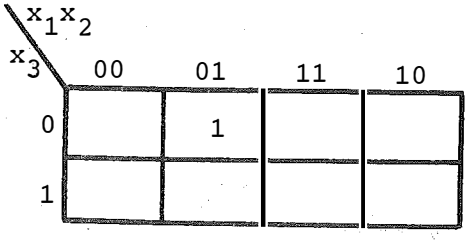
$$f = \bar{x}_1 . x_2 + x_1 . x_2 = (\bar{x}_1 + x_1)\ x_2 = x_2$$

se puede simplificar mucho. En este caso de sólo dos variables es sencillo, pero en cuanto sube un poco el número de varia-- bles el problema se complica.

Existen dos tipos de métodos sistemáticos de simplificación: los gráficos y los numéricos. Los primeros son más sencillos, pero en cuanto aumenta el número de variables se hacen inoperantes. Los numéricos en cambio permiten minimizar funciones con cualquier número de variables y ser programados para su utilización en el ordenador; el más conocido de estos últimos es el llamado método de Quine McCluskey, en el que no entraremos, limitándonos a exponer el más conocido de los métodos gráficos: el de Karnaugh.

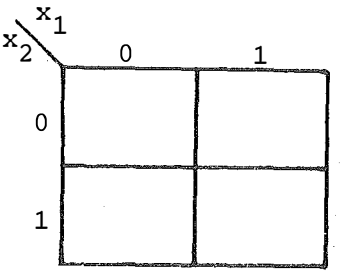
El método de Karnaugh se basa en la colocación de los valores en forma de tabla de doble entrada situando los elementos que se pueden simplificar por distributividad en posiciones contiguas de forma que sean fácilmente agrupables. Se utiliza para funciones de hasta cinco variables y la forma de construir estos mapas es la representada en la página siguiente.

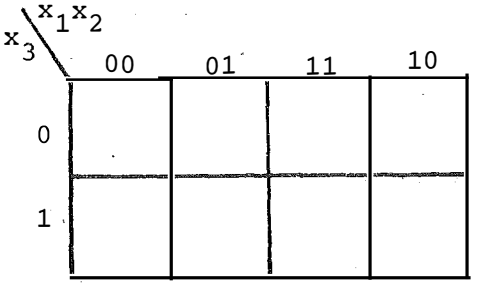
Las tablas se rellenan situando el valor que le corresponde a cada n-tupla en el cuadrado que le corresponde componiendo los elementos de ambos lados. Si estamos en n = 3 y el valor de la n-tupla ($x_1 = 0$, $x_2 = 1$, $x_3 = 0$) es 1, en el cuadrado se situaría

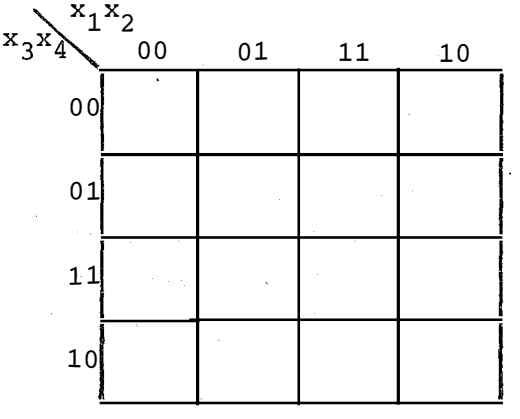


Cuando n = 5 según $x_5 = 1$ ó $x_5 = 0$, se sitúa en un cuadro o en otro.

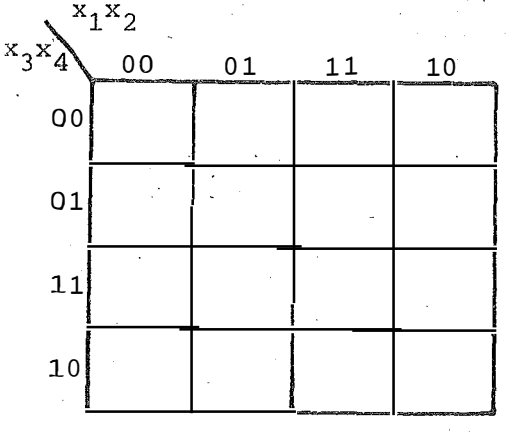
Una vez llena la tabla, los elementos contiguos tienen formas canónicas que pueden ser simplificadas por distributividad. Esto se debe a la forma en que se ha construido la tabla


x_1
x_2
0
1
0
1

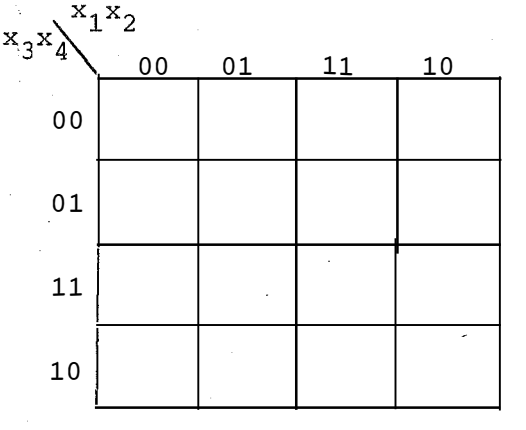

x_1x_2
x_3
00
01
11
10
0
1


x_1x_2
x_3x_4
00
01
11
10
00
01
11
10

$x_5 = 0$


x_1x_2
x_3x_4
00
01
11
10
00
01
11
10

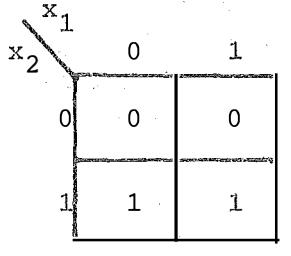
$x_5 = 1$


x_1x_2
x_3x_4
00
01
11
10
00
01
11
10

en que cada elemento y su contiguo sea horizontalmente o verti calmente, tienen productos canónicos que se diferencian únicamente en que una de las variables está negada en el otro. Los elementos de los lados también se corresponden con el simétrico en el lado opuesto. Cuando n = 5 además de estas correspondencias están las que hay entre elementos situados en las mismas posiciones.

Vamos a aclarar todo esto con algunos ejemplos:

Ejemplo 5.1. En el ejemplo $f = \bar{x}_1 . x_2 + x_1 . x_2$ de dos variables tenemos el siguiente mapa de Karnaugh:



Vemos que los dos unos están situados en posiciones contiguas. Esto nos permite eliminar x_1 por distributividad por lo que estos dos unos quedan determinados unicamente por el 1 de x_2. De aquí se deduce que $f = x_2$ obteniéndola ya simplificada de entrada.

El proceso y el resultado es idéntico al que realizamos para simplificarla cuando escribimos:

$$f = \bar{x}_1 . x_2 + x_1 . x_2 = (\bar{x}_1 + x_1)\ x_2 = x_2$$

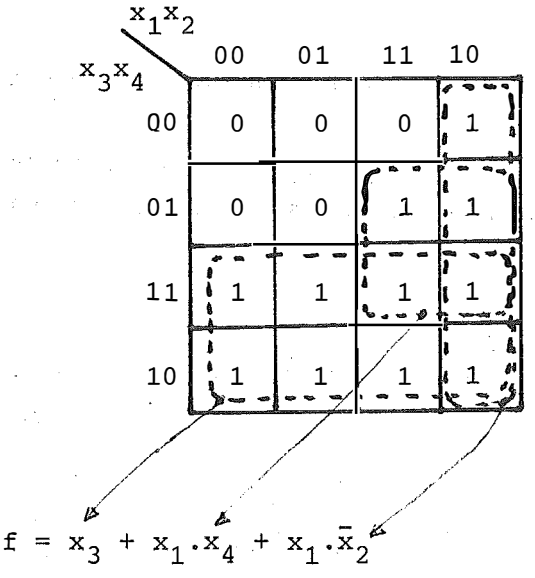
Ejemplo 5.2. Tenemos una función de 4 variables:

$$f = x_3 . x_4 . x_2 + x_1 . \bar{x}_2 + \bar{x}_2 . x_3 . x_1 + x_4 . x_1 + \bar{x}_1 . x_3 + x_2 . x_3 \ . \ x_4$$

Esta función es más dificil de simplificar; primero,

hallamos su tabla de valores, y después, sobre el mapa de Karnaugh buscamos agrupamientos:

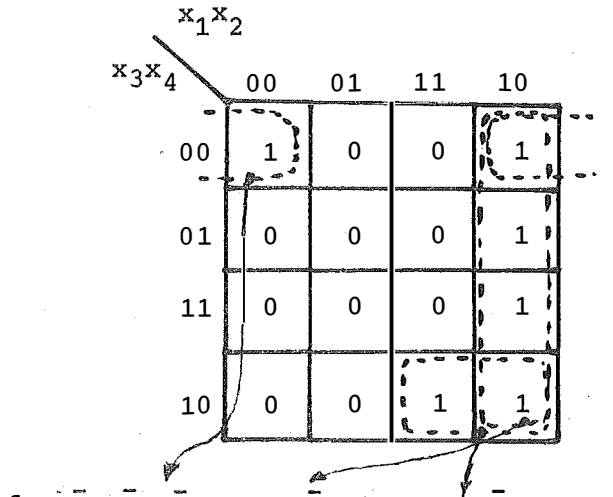
x_1	x_2	x_3	x_4	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



f (0000) = 0.1.0 + 0.1 + 1.0.0 + 0.0 + 1.0 + 0.0.0 = 0

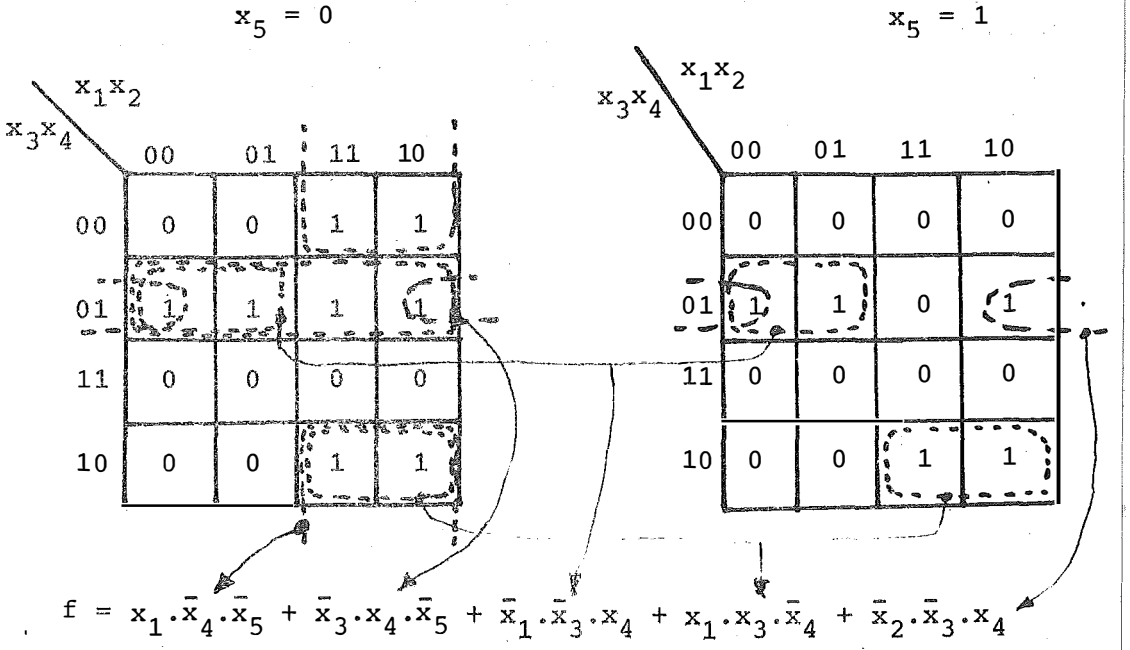
f (1111) = 1.0.1 + 1.0 + 0.1.1 + 1.1 + 0.1 + 1.1.1 = 1

Ejemplo 5.3. Damos directamente el mapa de Karnaugh:



$$f = \bar{x}_2.\bar{x}_3.\bar{x}_4 + x_1.\bar{x}_2 + x_1.x_3.\bar{x}_4$$

Ejemplo 5.4. Veamos la minimización de una función de cinco variables:



$$f = x_1.\bar{x}_4.\bar{x}_5 + \bar{x}_3.x_4.\bar{x}_5 + \bar{x}_1.\bar{x}_3.x_4 + x_1.x_3.\bar{x}_4 + \bar{x}_2.\bar{x}_3.x_4$$

Ejemplo 5.5.

Desarrollaremos ahora un ejemplo que, además de ser de aplicación práctica, nos permitirá ilustrar un caso interesante en la minimización de funciones de conmutación: el de las *funciones incompletamente especificadas*. Se trata de la realización de una etapa de sumador binario, es decir, un circuito que deberá sumar dos dígitos binarios, teniendo en cuenta el posible acarrero o arrastre de la suma de los dos dígitos de peso inmediatamente inferior (que pueden haberse sumado en otra etapa); por tanto, tendrá tres entradas binarias: x e y, correspondientes a los dígitos a sumar, y r', arrastre anterior, y dos salidas: s(suma) y r(arrastre producido). Un sumador paralelo de n bits constará de n etapas (aunque la primera no necesita la entrada r'), en las que la salida r de cada una se conecta a la entrada r' de la siguiente.

Si analizamos las distintas posibilidades de x, y, r' y para cada una anotamos los valores que deben tomar s y r, llegamos a la siguiente tabla de verdad (en realidad son dos: una para s y otra para r):

x	y	r'	s	r
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Las tablas de Karnaugh correspondientes son:

s

r \ xy	00	01	11	10
0	0	1	0	1
1	1	0	1	0

r'

r \ xy	00	01	11	10
0	0	0	1	0
1	0	1	1	1

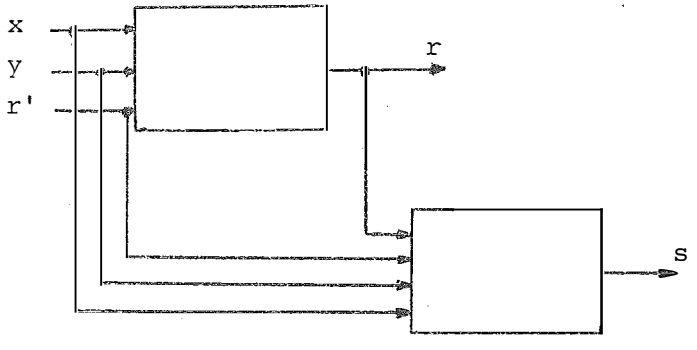
De las que resulta:

$$s = \bar{x}.\bar{y}.r' + \bar{x}.y.\bar{r}' + x.y.r' + x.\bar{y}.\bar{r}'$$

$$r = x.y + r'.(x+y)$$

(x.y es el "arrastre generado" en esta etapa, y r'.(x+y) es el "arrastre propagado" de la anterior).

Se observará que s no se ha podido simplificar, y se ha expresado por su forma canónica. Sin embargo, podemos reducir el número de puertas del circuito global si tenemos en cuenta que, en realidad, estamos diseñando dos circuitos: uno para s y otro para r; cabe entonces pensar en utilizar r como entrada adicional para s, que así será una función de cuatro entradas (x, y, r', r), como muestra la figura adjunta.



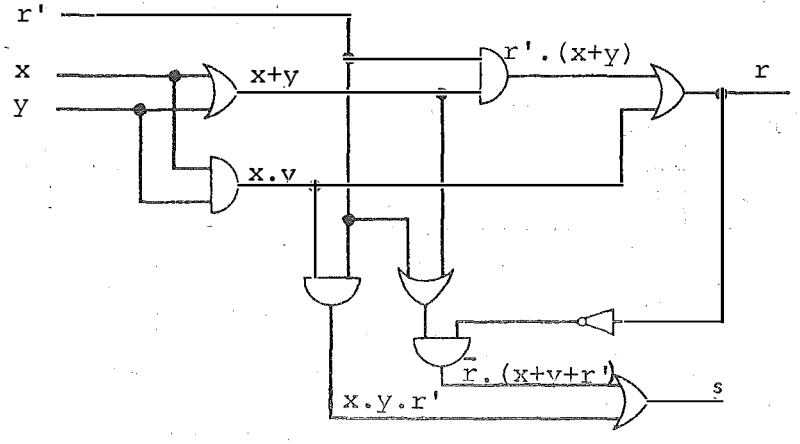
Si, siguiendo esta idea, tratamos a s como una función s = s (x,y,r',r), lo primero que encontraremos es que existen cuádruplas para las que s no está definida, por ejemplo, x =0, y = 0, r' = 0, r = 1; y s no está definida en este caso (y en otros similares) sencillamente porque esa combinación es imposible, ya que si x, y, r' valen las tres 0, entonces r = 0 necesariamente. Para estas cuádruplas de entrada, que no se van a presentar nunca a la entrada del circuito que da s podemos tomar, a efectos de minimizar, el valor 0 o el valor 1 indiferentemente para s, y esto lo representamos en la tabla de Karnaugh con el símbolo "Ø". Agrupando así las Ø's con los 1's o no, según convenga, llegamos a la forma mínima para s:

s \ xy / r'r	00	01	11	10
00	0	1	Ø	1
01	Ø	Ø	0	Ø
11	Ø	0	1	0
10	1	Ø	Ø	Ø

$$s = r'.\bar{r}+y.\bar{r}+x.\bar{r}+x.y.r' =$$

$$= \bar{r}.(x+y+r')+x.y.r'$$

El circuito total resultante para la etapa de sumador, utilizando unos elementos (puertas), que definiremos en el siguiente Capítulo, es:



CAPITULO 6.

REPRESENTACION DE FUNCIONES DE CONMUTACION. OTRAS FORMAS CANONICAS Y FORMAS MINIMAS

1. INTRODUCCIÓN.

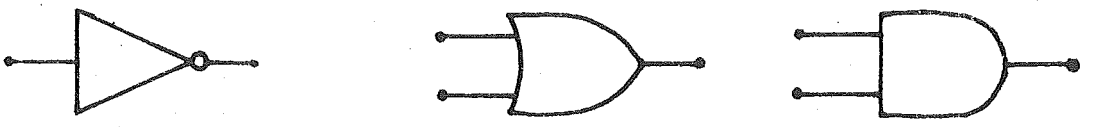
El presente capítulo se va a dedicar al estudio de las formas de representación gráfica de las funciones de conmutación. Esta representación gráfica es de gran importancia porque es directamente traducible a un circuito electrónico. La representación se hace del siguiente modo: Tenemos unas entradas que representan las variables. En estas entradas podemos colocar nosotros los valores que queramos. Las operaciones se realizan en una caja donde entran los elementos del álgebra que vamos a operar y a la salida obtenemos el elemento resultante. La importancia de este método está en que estas cajitas, llamadas en general, puertas, se pueden construir electrónicamente. Los "ceros y "unos" se pueden simular con un voltaje 0 y V respectivamente (por ejemplo). Si enlazamos las puertas como indica la representación gráfica obtenemos un circuito electrónico equivalente a nuestra forma booleana. Para cada n-tupla simulada electrónicamente obtenemos a la salida el valor correspondiente en la tabla de verdad. La constitución interna de las puertas se estudia en la asignatura de Electrónica digital. Aquí nos limitaremos a definir funcionalmente los principales tipos y a ver cómo pueden interconectarse para realizar físicamente los circuitos de conmutación.

2. LAS PUERTAS.

Una operación se puede describir como un proceso, donde entran en una caja los elementos que se van a operar y de donde sale el elemento resultante. Esto es lo que se hace en la representación. La negación tendrá una entrada y una salida; la suma y el producto tendrán dos entradas y una salida.

Las representaciones más utilizadas son:

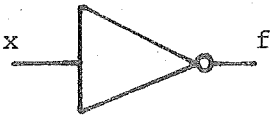
negación	suma	producto
N O T	O R	A N D



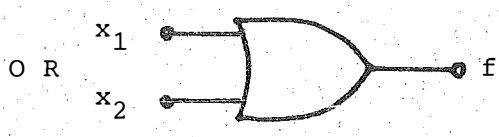
Las denominaciones NOT, OR y AND corresponden al inglés y es la forma en que se las denomina habitualmente en la literatura técnica.

La función que realizan estas puertas es la equivalente a la operación que representan. Sus tablas de verdad son, por tanto, las ya definidas anteriormente:

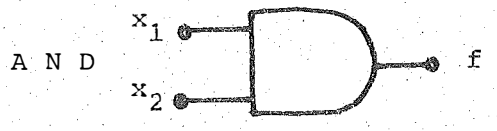
NOT



x	f
0	1
1	0



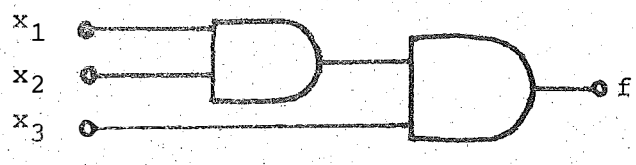
x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1



x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

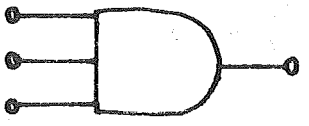
Las operaciones, en principio, son siempre entre dos entradas:

$f = (x_1 . x_2) . x_3$ sería



pero como el producto es asociativo, podemos representar esto

mediante una puerta de 3 entradas.



Así, existen puertas de cualquier número de entradas - para el producto. Con la suma podemos hacer lo mismo.

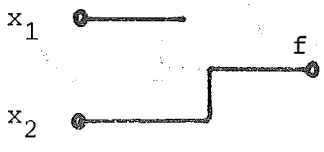
3. REPRESENTACION DE FORMAS MEDIANTE PUERTAS.

Una forma booleana es un conjunto de variables opera-- das entre si de una forma determinada. La representación ten-- drá por tanto unas entradas que representarán las variables a las que nosotros asignaremos valores según la función de asig- nación α. Estas irán operándose en las puertas según se indica en la forma booleana.

Vamos a ver los ejemplos del Capítulo 5.

Ejemplo 3.1.

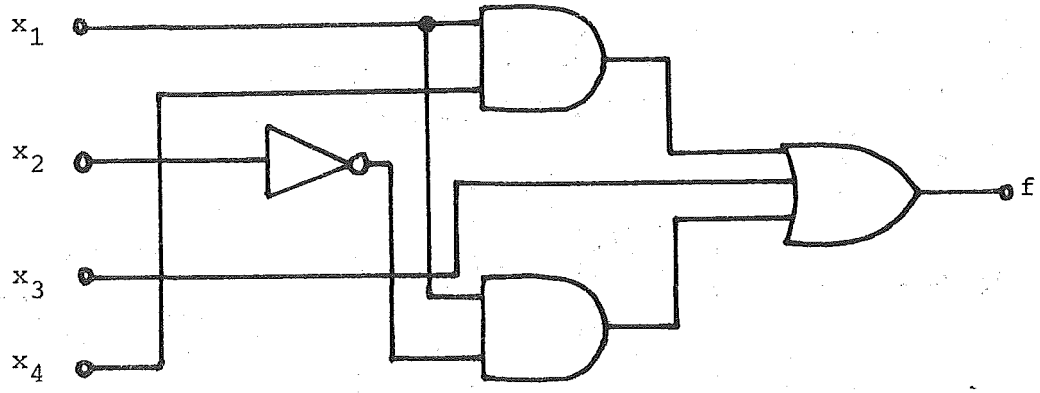
$f = x_2$



no se realiza ninguna operación por lo que la salida será - - igual a x_2

Ejemplo. 3.2.

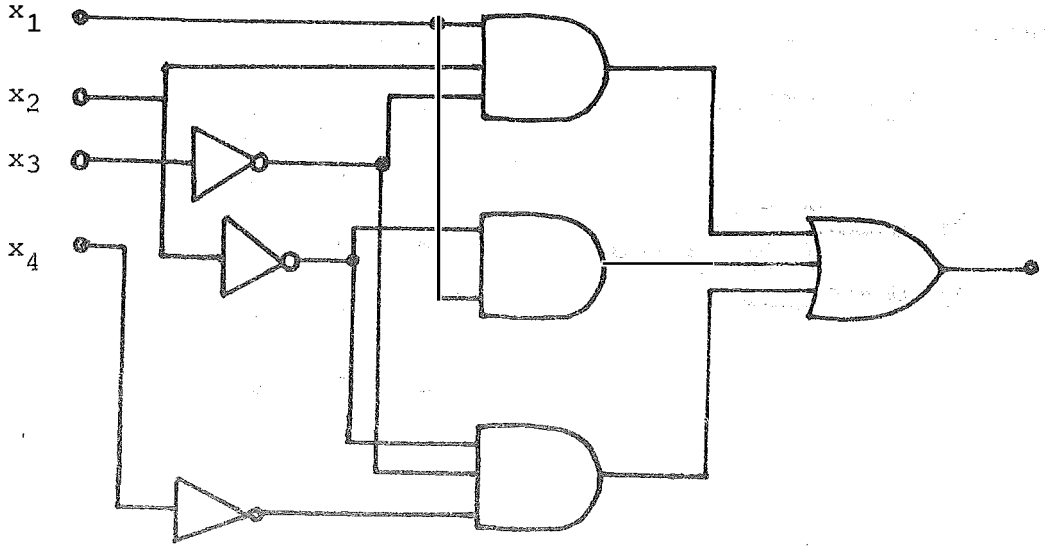
$$f = x_3 + x_1 . x_4 + x_1 . \bar{x}_2$$



Como se puede apreciar la forma booleana se simula como un proceso donde entran los valores de las variables de entrada. Estas se van operando según se indica y una vez hechas todas las operaciones obtenemos el valor de la función de valuación correspondiente.

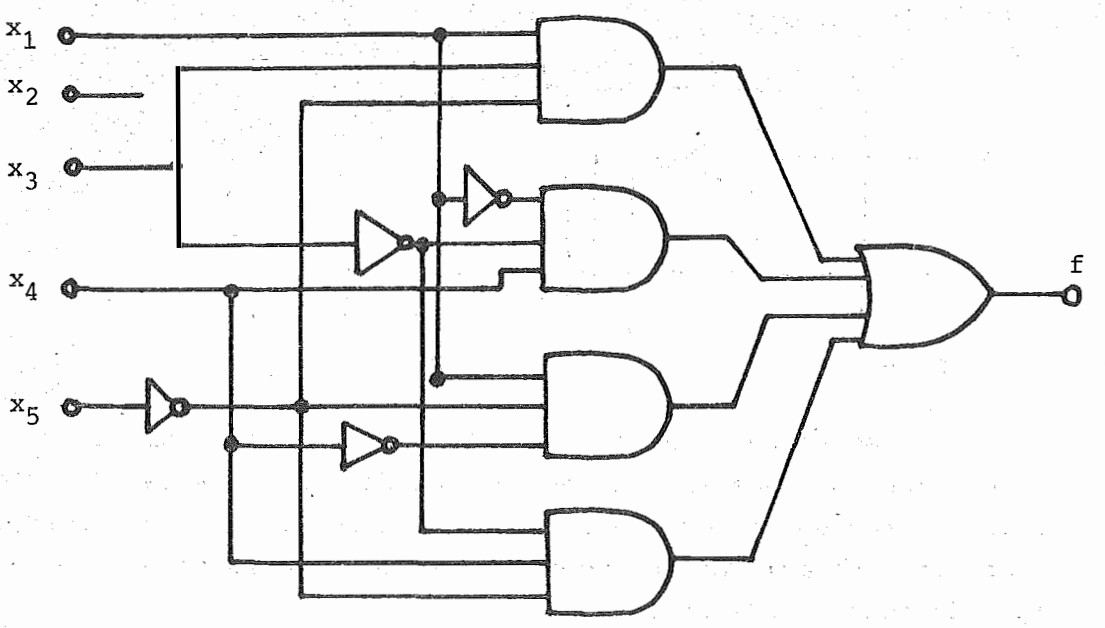
Ejemplo 3.3.

$$f = \bar{x}_2 . \bar{x}_3 . \bar{x}_4 + x_1 . \bar{x}_2 + x_1 . x_2 . \bar{x}_3$$



Ejemplo 3.4.

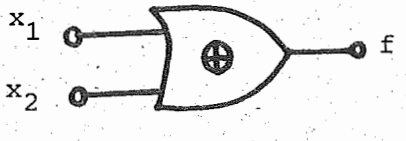
$$f = x_1.\bar{x}_4.\bar{x}_5 + \bar{x}_3.x_4.\bar{x}_5 + \bar{x}_1.\bar{x}_3.x_4 + x_1.x_3.\bar{x}_5$$



4. OTRAS PUERTAS.

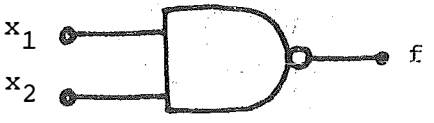
Además de estas tres puertas, existen en la práctica - otras que pueden simplificar mucho la representación gráfica y, consiguientemente, reducir el coste del circuito. Las más importante son las siguientes:

Ø R EXCLUSIVO



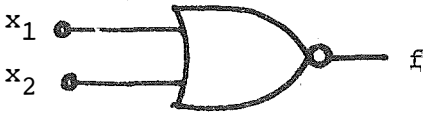
x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	0

N A N D



x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

N Ø R



x_1	x_2	f
0	0	1
0	1	0
1	0	0
1	1	0

Estas, con las tres anteriores, nos dan un conjunto de puertas que flexibilizan mucho la representación gráfica. En realidad se pueden construir unas con otras:

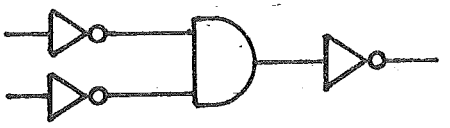
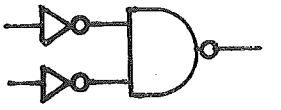
Ya hablamos en capítulos anteriores sobre la posibilidad de representar conectivas en un caso, operaciones en otro y puertas ahora en función de otras.

Las leyes de de Morgan nos dicen que:

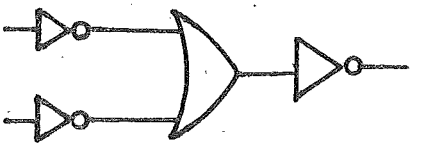
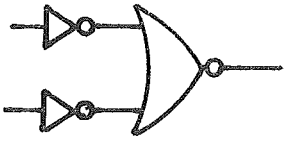
$$x_1 + x_2 = \overline{\bar{x}_1 . \bar{x}_2}$$

$$x_1 . x_2 = \overline{\bar{x}_1 + \bar{x}_2}$$

Apoyándonos en estas identidades podemos realizar la - operación de suma lógica con puertas AND y NOT:

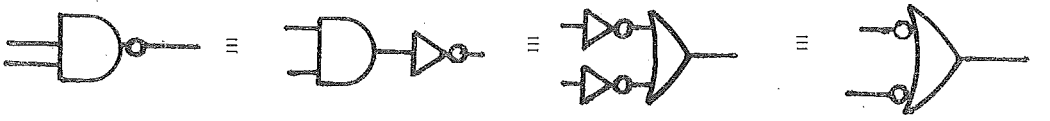
 o bien 

y el producto con puertas OR y NOT:

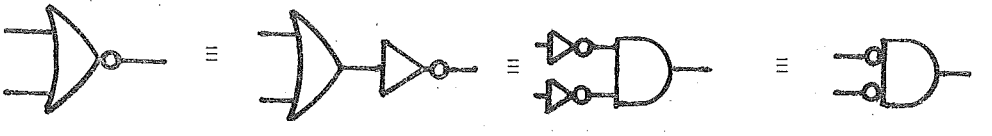
 o bien 

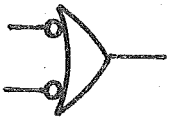
Por otra parte, las representaciones de NAND, NOR y OR exclusivo con puertas AND, OR y NOT serían:

NAND: $f = \overline{x_1 \cdot x_2} = \bar{x}_1 + \bar{x}_2$

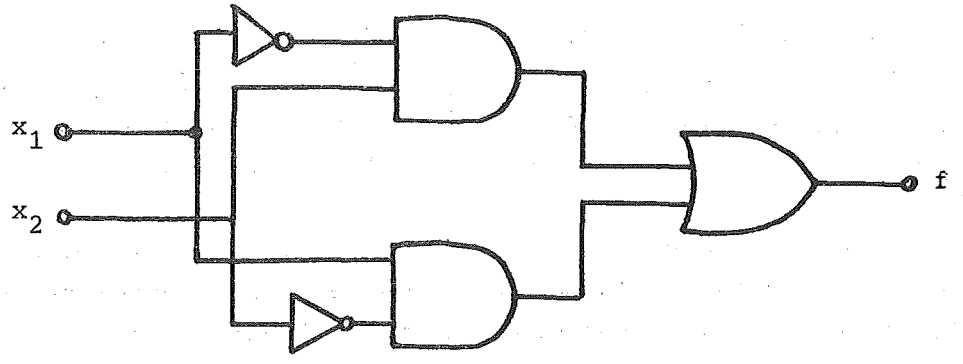


NOR: $f = \overline{x_1 + x_2} = \bar{x}_1 \cdot \bar{x}_2$



(Nótense las representaciones alternativas para las - puertas NAND y NOR:  para NAND y  para NOR).

Ø R EXCLUSIVO



$$f = x_1.\bar{x}_2 + \bar{x}_1.x_2$$

5. LA SEGUNDA FORMA CANÓNICA Y LA FORMA MÍNIMA EN PRODUCTO DE SUMAS.

5.1. La segunda forma canónica.

En el Capítulo 4 vimos cómo cada clase de equivalencia en el conjunto de formas booleanas que pueden construirse con n variables puede representarse por una forma canónica formada por sumas de productos canónicos; existen, como vamos a ver, otras posibles formas booleanas que pueden utilizarse como formas canónicas, por lo que, para diferenciarlas, llamaremos a la que ya conocemos primera forma canónica o forma canónica en suma de productos. A los productos canónicos a veces se les llama minitérminos.

Análogamente a como definíamos el producto canónico (Def. 4.2.2.1 del Cap. 4), podemos definir la suma canónica o maxitérmino, S_n, de las variables $x_1,....x_n$ como cualquier forma booleana del tipo

$$S_n = l_1 + l_2 + ... + l_n = \sum_{1}^{n} l_i$$

(Σ representa la suma booleana; recuérdese que l_i representa - x_i o $\bar{x}_i$). La segunda forma canónica o forma canónica en producto de sumas se define como una forma booleana compuesta por - productos de sumas canónicas diferentes entre sí.

Siguiendo un proceso paralelo al del Cap. 4, puede demostrarse que las sumas canónicas son elementos atómicos de - $<B_n, +, ., ->$ y que cada clase de equivalencia en el conjunto de formas booleanas de n variables puede representarse por una forma canónica en producto de sumas.

Pasando ya al caso de mayor trascendencia práctica, el de las funciones de conmutación, el problema más inmediato a - resolver es el de cómo obtener la segunda forma canónica co- - rrespondiente a una función a partir de su tabla de verdad (el problema inverso, como en el caso de la primera forma canónica, es fácil: hasta obtener la función de valuación para cada una de las posibles n-tuplas). Si la función es f, como $f + \bar{f} = 1$, si sumamos los productos canónicos correspondientes no a los - "unos", sino a los "ceros" de la tabla, entonces se obtiene la primera forma canónica correspondiente a $\bar{f}$, y, aplicando los - dos teoremas de de Morgan llegamos a la forma en producto de - sumas:

$$\bar{f} = \sum_{1}^{n} \left(\prod_{1}^{n} l_i \right);$$

$$f = \prod_{1}^{\overline{n}} \left(\prod_{1}^{n} l_i \right) = \prod_{1}^{n} \left(\sum_{1}^{n} \bar{l}_i \right)$$

$$f = x_3 + x_1.x_4 + x_1.\bar{x}_2$$

Si, sobre la misma tabla de Karnaugh, en lugar de agrupar los "unos" agrupamos los "ceros" resulta:

$$\bar{f} = \bar{x}_1.\bar{x}_3 + x_2.\bar{x}_3.\bar{x}_4$$

Por lo que

$$f = (x_1 + x_3).(\bar{x}_2 + x_3 + x_4)$$

La realización de esta función en esta forma requiere una puerta NOT, una AND y dos OR. En este caso, el coste de la realización es prácticamente igual al de la realización en base a la forma en suma de productos (Ejemplo 3.2 de este Capítulo).

Invitamos al lector a minimizar en esta forma los Ejemplos 5.1 a 5.4 del Capítulo 5, dibujar los circuitos resultantes y compararlos con los obtenidos por suma de productos.

6. Formas con sólo operaciones NAND y con sólo operaciones NOR.

6.1. NAND y NOR son operaciones completas.

Anteriormente hemos visto dos tipos de puertas que, por motivos tecnológicos (facilidad de integración de los circuitos) son muy utilizadas: las puertas NAND y NOR. Por la frecuencia de su utilización se recurre a símbolos especiales para representar las funciones realizadas por estas puertas: "/" para NAND y "↓" para NOR. Es decir:

$$x/y/z/\ldots = \overline{x.y.z.\ldots} = \bar{x}+\bar{y}+\bar{z}+\ldots$$

$$x{\downarrow}y{\downarrow}z{\downarrow}\ldots = \overline{x+y+z+\ldots} = \bar{x}.\bar{y}.\bar{z}\ldots$$

Tanto NAND como NOR tienen la propiedad de ser operaciones completas; esto quiere decir que cualquier función de conmutación puede representarse utilizando sólo la operación NAND o utilizando sólo la operación NOR. Para demostrarlo basta con ver que las tres operaciones básicas, complementación, producto y suma, pueden realizarse con ellas:

a) $\bar{x} = \overline{x.x} = x/x$

$\bar{x} = \overline{x+x} = x\downarrow x$

b) $x.y = \overline{\overline{x.y}} = \overline{x/y} = (x/y)/(x/y)$

$x.y = \overline{\bar{x}+\bar{y}} = \bar{x}\downarrow\bar{y} = (x\downarrow x)\downarrow(y\downarrow y)$

c) $x+y = \overline{\bar{x}.\bar{y}} = \bar{x}/\bar{y} = (x/x)/(y/y)$

$x+y = \overline{x+y} = (\overline{x\downarrow y}) = (x\downarrow y)\ (x\downarrow y)$

Es preciso prestar atención a los paréntesis, ya que, a diferencia de la suma y el producto, NAND y NOR no son asociativas:

$$(x/y)/z \neq x/(y/z) \neq x/y/z$$

$$(x\downarrow y)\downarrow z \neq x\downarrow(y\downarrow z) \neq x\downarrow y\downarrow z$$

(Compruébese por medio de las respectivas tablas de verdad).

6.2. La tercera forma canónica.

La tercera forma canónica (sólo con NAND) se obtiene directamente de la primera mediante aplicación del

Teorema 6.2.1:

$$(x_1.x_2.\ldots.x_n) + (y_1.y_2.\ldots.y_m) + \ldots + (z_1.z_2.\ldots.z_h) =$$

$$= (x_1/x_2/\ldots/x_n)\ /\ (y_1/y_2/\ldots/y_m)\ /\ldots/\ (z_1/z_2/\ldots z_h)$$

(Para demostrarlo basta con aplicar dos veces las leyes de de Morgan).

6.3. La forma mínima sólo con NAND.

Se obtiene aplicando el Teorema 6.2.1 a la forma mínima en suma de productos.

Ejemplo. (Ejemplo 5.2 del Capítulo 5):

$$f = x_3+x_1.x_4+x_1.\bar{x}_2 = (x_3/x_3)/(x_1/x_4)/(x_1/\bar{x}_2) =$$

$$= (x_3/x_3)/(x_1/x_4)/(x_1/(x_2/x_2))$$

(Obsérvese que cuando uno de los productos consta de una sola variable, en este caso x_3, este producto puede representarse como $x_3.x_3$, y de ahí que al aplicar el teorema pongamos x_3/x_3 (es decir, $\bar{x}_3$), y no x_3).

6.4. La cuarta forma canónica.

Se obtiene de la segunda aplicando el

Teorema 6.4.1:

$$(x_1+x_2+\ldots+x_n).(y_1+y_2+\ldots+y_m).\ldots.(z_1+z_2+\ldots z_h) =$$

$$= (x_1\downarrow x_2\downarrow\ldots\downarrow x_n)\downarrow(y_1\downarrow y_2\downarrow\ldots\downarrow y_m)\downarrow\ldots\downarrow(z_1\downarrow z_2\downarrow\ldots z_h)$$

(Se demuestra igualmente con las leyes de de Morgan)

6.5. La forma mínima sólo con NOR.

Si aplicamos el Teorema 6.4.1 a la forma mínima en producto de sumas, resultará una forma mínima con operaciones NOR solamente.

Ejemplo.

$$f = x.(\bar{x} + y).(y + \bar{z}) =$$

$$= (x \downarrow x) \downarrow (\bar{x} \downarrow y) \downarrow (y \downarrow \bar{z}) =$$

$$= (x \downarrow x) \downarrow ((x \downarrow x) \downarrow y) \downarrow (y \downarrow (z \downarrow z))$$

(Aquí podemos hacer una observación similar a la del ejemplo anterior: si una de las sumas sólo tiene un sumando, x en este caso, como $x = x+x$, al aplicar el Teorema ponemos $x \downarrow x = \bar{x}$, en lugar de x).

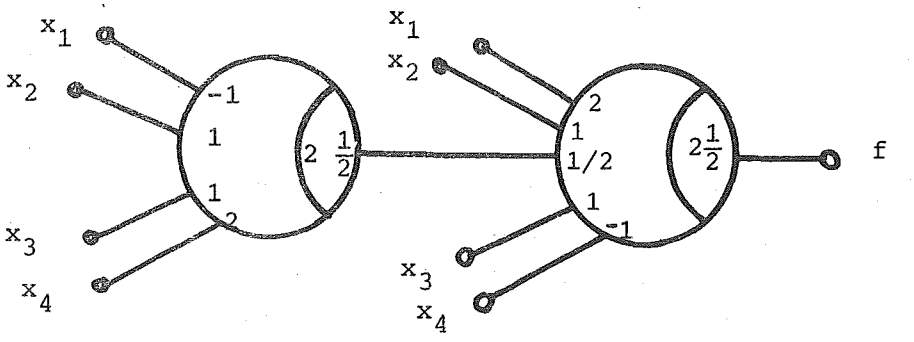
7. LÓGICA DE UMBRAL.

Otra forma de representación de funciones de conmutación que tiene interés en ciertas aplicaciones, es la que se realiza mediante la llamada lógica de umbral.

Esta lógica tiene una filosofía de diseño algo diferente, aunque los resultados obtenidos son idénticos a los de la representación con puertas NOT, AND y OR. El elemento básico que se utiliza aquí es el elemento o puerta de umbral (en inglés *threshold element*), que, en función de las entradas, a las cuales aplicamos "1" ó "0", obtenemos un "1" o un "0" a la salida. Cada entrada lleva asociado un cierto coeficiente de ponderación y cuando la suma ponderada de las entradas sobrepasa un cierto umbral, la salida toma el estado 1. De ahí proviene su nombre.

7.1. La puerta de umbral.

Supongamos que hay n entradas cuyos valores se representan por x_i y una salida cuyo valor se representa por y. Si ω_i es el coeficiente asociado a la entrada x_i, las ecuaciones



La forma de agrupar los unos de salida se indica en la figura con la línea de trazos. Como se ve es una realización mucho más sencilla que con puertas AND y OR. No son necesarios los inversores.

8. EJEMPLO DE APLICACIÓN.

Se desea diseñar un sistema de regulación de la temperatura y del nivel de agua en una piscina. El sistema recibirá la información de tres sensores colocados dentro de la piscina: un termómetro y dos detectores de nivel, y deberá actuar sobre una resistencia para calentar el agua y sobre dos válvulas: una que permite desalojar agua, y otra que permite añadir agua fría. Las especificaciones de su funcionamiento vienen dadas por la siguiente tabla:

Información recibida		Acción a tomar
Temperatura	Nivel	
Normal	Normal	Ninguna
Normal	Bajo	Añadir agua
Normal	Alto	Sacar agua
Baja	Normal	Calentar
Alta	Normal	Añadir agua
Baja	Bajo	Añadir agua y calentar
Baja	Alto	Sacar agua y calentar
Alta	Bajo	Añadir agua
Alta	Alto	Añadir y sacar agua al mismo tiempo.

Se supondrá que el termómetro tiene dos terminales de salida, en cada uno de los cuales pueden tenerse dos niveles de tensión (a los que daremos los niveles lógicos 0 y 1). Si la temperatura es demasiado alta, el primer terminal, T_1, estará en el nivel 1 (y el segundo, T_2, en el 0); si es demasiado baja ocurrirá lo contrario, y si está entre los límites prefijados ("temperatura normal"), ambos serán 0.

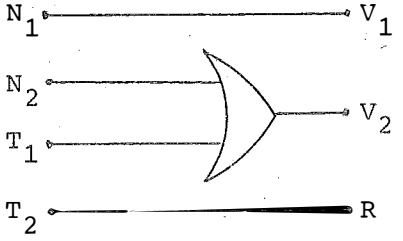
Los detectores del nivel de agua dan también señales 0 y 1: uno de ellos, N_1, está en 1 si el nivel es demasiado alto, y el otro, N_2, está en 1 si es demasiado bajo (de modo que ambos en 0 indicarán "nivel normal").

En la salida, para que una válvula (V_1: sacar, V_2: añadir) esté cerrada se deberá dar el nivel lógico 0, y para que esté abierta el 1. Análogamente, para que la resistencia (R) caliente se dará un 1 y para que no actúe un 0.

La parte lógica del sistema será un circuito con cuatro entradas (T_1,T_2,N_1,N_2) y tres salidas (V_1,V_2,R). Las tablas de verdad se obtienen inmediatamente de las especificaciones. Se observará que hay 7 combinaciones de las variables de entrada que son imposibles (las que corresponden a $T_1=T_2=1$ y $N_1=N_2=1$), por lo que las funciones V_1,V_2 y R son incompletamente especificadas. Aprovecharemos este hecho, del mismo modo que en el Ejemplo 5.5 del Capítulo anterior, para minimizar tales funciones. Si el lector sigue los pasos necesarios llegará al siguiente resultado:

$$V_1 = N_1;\ V_2 = T_1 + N_2;\ R = T_2,$$

por lo que el circuito será, simplemente:



CAPITULO 7.

LOGICA PROPOSICIONAL Y ALGEBRA DE BOOLE

1. INTRODUCCIÓN.

Hemos hablado anteriormente de la estructura de álgebras de Boole que poseen ciertos conjuntos dotados de las oportunas operaciones. El conjunto S de todas las sentencias, el conjunto {0,1} sobre el que interpretábamos éstas, el conjunto de clases de un conjunto, etc., son ejemplos típicos de estas estructuras. De las aplicaciones del álgebra de Boole sólo hemos visto las referentes al diseño lógico. Aquí vamos a ver la interrelación existente entre la lógica proposicional y las estructuras algebráicas.

2. ESTRUCTURA ALGEBRÁICA DE LA LÓGICA PROPOSICIONAL.

Como primer paso para el estudio de la estructura de la lógica proposicional, vamos a estudiar el conjunto de las sentencias algebraicamente. Una vez que hayamos analizado bien la sintaxis pasaremos a estudiar la estructura algebráica de las interpretaciones que realizamos de estas sentencias, o lo que es lo mismo, de la semántica.

2.1. Equivalencia entre sentencias.

Cuando establecimos el alfabeto y la sintaxis de la lógica proposicional dimos sólo las reglas y propiedades imprescindibles para la formación de sentencias, pero nada más.

Aquí nos centraremos en el estudio del conjunto S de las sentencias. (Seguiremos utilizando la notación del capítu-

lo 2). Dentro de S se puede establecer una relación de orden - parcial, definida del siguiente modo.

$$A \vdash B \text{ si y sólo si } (A \to B) \in T$$

Obsérvese que aquí el símbolo $\vdash$ no significa demostrabilidad, sino deducibilidad de B en función de A. En función del contexto se pueden diferenciar los significados en ambos casos.

En función de la deducibilidad se puede definir una relación de equivalencia entre sentencias diciendo que A y B son equivalentes si cada una de ellas puede deducirse de la otra:

$$A \vdash B \quad y \quad B \vdash A$$

que en notación formalizada sería:

$$((A \to B) \wedge (B \to A)) \in T$$

que, como más adelante se verá, equivale a:

$$(A \leftrightarrow B) \in T$$

Esta relación la representaremos de ahora en adelante como

$$A \equiv B$$

Teorema 2.1.1.

Si $A \equiv A'$ y $B \equiv B'$ entonces

$$\neg A \equiv \neg A'$$

$$A \wedge B \equiv A' \wedge B'$$

$$A \vee B \equiv A' \vee B'$$

$$A \rightarrow B \equiv A' \rightarrow B'$$

$$A \leftrightarrow B \equiv A' \leftrightarrow B'$$

(La demostración es complicada y requiere múltiples conceptos intermedios. El lector interesado puede encontrarla en la bibliografía).

La importancia de este teorema radica en lo siguiente: igual que cuando definimos las formas booleanas establecimos una relación de equivalencia y por tanto una partición, aquí hemos definido unas clases de equivalencia también que además son operables entre si. Existirán múltiples sentencias equivalentes, pero ya no las trataremos separadamente sino en función de la clase a la que pertenecen. Las clases las designamos por α, β,....., γ. Este conjunto se suele designar por S/R. Entre este conjunto y S podemos definir la siguiente aplicación T:

$$\neg\alpha = T(\neg A)$$

$$\alpha \wedge \beta = T(A \wedge B)$$

$$\alpha \vee \beta = T(A \vee B)$$

$$\alpha \rightarrow \beta = T(A \rightarrow B)$$

$$\alpha \leftrightarrow \beta = T(A \leftrightarrow B)$$

Esta relación establece una correspondencia entre clases y los elementos que la componen. El conjunto T constituye una clase de equivalencia que representamos por 1. El conjunto formado por todas las sentencias de la forma (¬ A) tal que - A ∈ T constituye otra clase de equivalencia que representare-- mos por 0.

Se puede verificar que

$$\left.\begin{array}{l} \alpha \wedge \beta = \beta \wedge \alpha \\ \alpha \vee \beta = \beta \vee \alpha \end{array}\right\} \text{ conmutatividad}$$

$$\left.\begin{array}{l} \alpha \wedge (\beta \wedge \gamma) = (\alpha \wedge \beta) \wedge \gamma \\ \alpha \vee (\beta \vee \gamma) = (\alpha \vee \beta) \vee \gamma \end{array}\right\} \text{asociatividad}$$

$$\left.\begin{array}{l} \alpha \wedge (\beta \vee \gamma) = (\alpha \wedge \beta) \vee (\alpha \wedge \gamma) \\ \alpha \vee (\beta \wedge \gamma) = (\alpha \vee \beta) \wedge (\alpha \vee \gamma) \end{array}\right\} \text{distributividad}$$

$$\left.\begin{array}{l} \alpha \vee \alpha = \alpha \\ \alpha \wedge \alpha = \alpha \end{array}\right\} \text{ idempotencia}$$

$$\alpha \vee \neg\alpha = 1$$

$$\alpha \vee 1 = 1$$

$$\alpha \wedge 0 = 0$$

$$\alpha \wedge \neg\alpha = 0$$

$$\alpha \wedge 1 = \alpha$$

Es decir que S/R tiene estructura de álgebra de Boole con respecto a "$\wedge$", "$\vee$" y "$\neg$". El resto de las conectivas - son función de estas tres primeras:

$$\alpha \rightarrow \beta = \neg\alpha \vee \beta$$

$$\alpha \leftrightarrow \beta = (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha).$$

También existe la posibilidad de expresar "$\vee$" en función de "$\neg$" y "$\wedge$" y "$\wedge$" en función de "$\neg$" y "$\vee$":

$$\alpha \vee \beta = \neg(\neg\alpha \wedge \neg\beta)$$

$$\alpha \wedge \beta = \neg(\neg\alpha \vee \neg\beta)$$

La demostración se realizaría comprobando la equivalencia entre

$$u \rightarrow v \equiv \neg u \vee v$$

$$u \leftrightarrow v \equiv (u \rightarrow v) \wedge (v \rightarrow u)$$

$$u \vee v \equiv \neg(\neg u \wedge \neg v)$$

$$u \wedge v \equiv \neg(\neg u \vee \neg v)$$

bien mediante una demostración formal o con sus tablas de verdad.

2.2. Interpretación de sentencias.

La interpretación de sentencias se hace mediante una aplicación entre nuestras variables y el conjunto {0,1}. Es equivalente a asignar valores a las variables proposicionales que estamos manejando. Las sentencias y las formas booleanas ya hemos visto que son idénticas. La interpretación también se puede asemejar con la función de asignación que definimos para las formas. Sintácticamente son estructuras idénticas. Todavia

más, la extensión de la interpretación de variables se corresponde con la función de valuación de una forma establecida en base a su función de valuación.

La diferencia entre la interpretación de sentencias y la valuación de formas radica en lo siguiente: el cálculo proposicional clásico establece sólo la verdad o la falsedad de sentencias, que en términos matemáticos significa la interpretación de una sentencia como un "0" o un "1". Nos deberemos mover por tanto en el ambiente de las álgebras de Boole binarias. Las formas booleanas son en cambio entes matemáticos definidos de forma totalmente general y que pueden ser aplicados a cualquier tipo de álgebra booleana. Cuando definimos las sentencias e interpretaciones, la hicimos de forma totalmente general también. Sus posibilidades de aplicación se hacen asi mucho más amplias pudiendo utilizarse en otro tipo de lógicas como las que estudiamos en el capítulo siguiente.

CAPITULO 8.

EXTENSIONES DE LA LOGICA CLASICA

1. INTRODUCCIÓN.

Aparte de la lógica de predicados, cuyo estudio no hemos abordado en este Tema, como ya decíamos al principio, y que suele considerarse parte integrante de la lógica clásica, se han producido extensiones o generalizaciones de ésta, principalmente en cuatro direcciones:

a) Introduciendo una gama más amplia de interpretaciones que la binaria "verdadero" o falso". De hecho, solemos utilizar habitualmente valoraciones sobre un juicio, tales como "es posible", "es bastante factible", "es poco probable", etc. Formalmente, y tal como hemos definido la interpretación semántica (Apartado 2.1 del Cap.2), esto es, al menos en principio, fácil: basta con considerar que el conjunto V sobre el que se interpreta no es $V = \{0,1\}$, sino que contiene un número mayor de elementos, que puede ser incluso infinito: $V = \{x \mid x \in [0,1]\}$. A partir de esta idea surgen las *lógicas polivalentes*, y al relacionarlas con el concepto de probabilidad aparece la *lógica probabilista*.

b) Introduciendo operadores que reflejen conceptos de posibilidad o necesidad ("no es necesario que", "es suficiente que", "es imposible que", etc.). Aparecen así las *lógicas modales*.

c) Suprimiendo uno o más axiomas (entre los que pare--

cen menos evidentes) del conjunto clásico, lo que - da lugar a las *lógicas debilitadas*. Obsérvese que - cualquier lógica no bivalente es también una lógica debilitada, ya que implica la anulación del principio clásico del tercio excluso (Con el esquema - de axiomas que dimos en el Apartado 1.3 del Cap. 2, este principio se deducía como teorema).

d) Sustituyendo el concepto clásico de "conjunto" por el de "conjunto borroso". Esto da lugar a la *lógica borrosa*, que puede considerarse como un tipo de lógica polivalente, pero que no es exactamente igual a la lógica probabilista.

De todas ellas, la que parece tener más posibilidades de aplicaciones técnicas en el futuro es la última. En este Capítulo nos limitaremos a decir unas palabras sobre lógicas polivalentes y a exponer muy sucintamente los principios de la - lógica borrosa.

2. LÓGICAS POLIVALENTES.

Ya hemos mencionado en algunos casos la poca flexibilidad de una interpretación sobre un conjunto de dos elementos. Aparte de la verdad o falsedad de una sentencia pueden existir grados intermedios de incertidumbre o de certeza.

Jan Lukasiewicz y Emil Post propusieron a principios - de siglo las llamadas lógicas polivalentes, para poder repre-- sentar posibilidades intermedias entre verdad y falsedad. El - número de posibilidades puede llegar a ser infinito. Su estu-- dio es complejo y precisa del desarrollo de un nuevo tipo de - álgebras, denominadas álgebras de Post.

Un caso particular de fácil comprensión es el de la ló

gica trivalente. La interpretación se hace aquí sobre un conjunto de tres elementos, {0,1/2,1}. El 0 representa la falsedad, el 1/2 la incertidumbre y el 1 la verdad (se supone que no hay grados de incertidumbre).

Las operaciones entre elementos se hacen según se indica en la siguiente tabla de verdad:

u	v	$\neg u$	$u \wedge v$	$u \vee v$	$u \rightarrow v$	$u \leftrightarrow v$
0	0	1	0	0	1	1
0	1/2	1	0	1/2	1	1/2
0	1	1	0	1	1	0
1/2	0	1/2	0	1/2	1/2	1/2
1/2	1/2	1/2	1/2	1/2	1	1
1/2	1	1/2	1/2	1	1	1/2
1	0	0	0	1	0	0
1	1/2	0	1/2	1	1/2	1/2
1	1	0	1	1	1	1

Puede comprobarse que si se eliminan todas las filas en las que aparece 1/2 resultan las tablas de verdad de la lógica binaria clásica.

Ejemplo. Recuérdese el ejemplo: "Si hay niebla no se ve", que se formalizaba: $u \rightarrow \neg v$ (Cap. 2). En lógica trivalente podemos tener, por ejemplo,la interpretación: u = 1, v = 1/2 (es decir, hay niebla, pero no sabemos si se ve o no); la interpretación de la sentencia sería: $(1 \rightarrow \neg 1/2) = (1 \rightarrow 1/2) = 1/2$, es decir, no está definido si la sentencia es verdadera o falsa (cosa lógica, ya que si, habiendo niebla, decimos que se ve, la sentencia es falsa: $1 \rightarrow \neg 1 = 1 \rightarrow 0 = 0$, mientras que si deci--

mos que no se ve, entonces es cierta: $1 \rightarrow \neg 0 = 1 \rightarrow 1 = 1$).

De una manera general, para lógicas que admiten un número cualquiera (finito o infinito) de valores de incertidumbre entre 0 (falso) y 1 (cierto), pueden definirse cuatro operadores básicos mediante las siguientes interpretaciones, llamadas "leyes de Lukasiewicz":

a) negación:

$$\neg u = 1-u$$

b) conjunción:

$$u \wedge v = \text{núm}\ (u,v)$$

c) disyunción:

$$u \vee v = \text{máx}\ (u,v)$$

d) condicional:

$$u \rightarrow v = 1, \quad \text{si } u \leq v$$

$$u \rightarrow v = 1-u+v, \quad \text{si } u > v$$

(El lector puede comprobar que estas leyes conducen a la misma tabla anterior en el caso trivalente).

Reichenbach, por los años 30, investigó las relaciones entre el cálculo de probabilidades y la lógica polivalente, iniciando así el estudio de la lógica probabilista. El operador de negación tiene una correspondencia inmediata en términos de probabilidades; así, por ejemplo, si tiramos un dado, en lenguaje de probabilidades podemos decir: *el suceso "sacar un tres" tiene probabilidad 1/6, y el suceso contrario tiene probabilidad 1-1/6 = 5/6*, mientras que en lenguaje lógico diríamos: *la proposición "al tirar un dado sale un tres" tiene el valor 1/6, y la proposición contraria tiene el valor 1-1/6 = 5/6*.

Los otros operadores lógicos ya presentan mayores dificultades para relacionarlos con las operaciones del cálculo de probabilidades.

3. LÓGICA BORROSA.

3.1. Introducción.

El concepto ya clásico de *conjunto* es el pilar fundamental de la Matemática, y nadie puede negar los servicios que ésta presta a las otras ciencias, proporcionándoles modelos y métodos de análisis de los mismos. Sin embargo, en la realidad se presentan situaciones (particularmente, cuando aparecen consideraciones subjetivas) en las que resulta difícil determinar la pertenencia o no de un elemento a un conjunto. Por ejemplo:

- el conjunto de los números naturales mucho mayores - que 100: parece evidente que 101 no pertenece al conjunto, y que 10^{10} si, pero ¿y 500?;
- el conjunto de las personas pobres: ¿pertenezco yo a ese conjunto?;
- el conjunto de las mujeres hermosas, etc.

Tales conjuntos pueden denominarse "borrosos" (o "difusos") para indicar que no existe un criterio que determine exactamente un límite entre pertenencia o no pertenencia al conjunto.

A esta altura, el lector atento puede objetar inmediatamente: "Lo que ocurre es que no se ha establecido el criterio para definir los conjuntos. Así, en el primer ejemplo, puede - decirse (por convenio, o por hipótesis de trabajo) que el límite está en 1000; en el segundo, que es pobre toda persona que, sin tener patrimonio, reciba unos ingresos inferiores al sala-

rio mínimo; en el último ejemplo ya es más difícil establecer un criterio,pero ¿a quién se le ocurre tratar matemáticamente conjuntos como ése?".

Ahora bien, si tratamos de formalizar las relaciones - del hombre con su entorno siempre vamos a encontrarnos con elementos imprecisos o "borrosos", sobre todo en la actividad más típicamente humana, el lenguaje. Cuando una persona está aprendiendo a conducir, en un determinado momento el instructor puede decirle: "levante ligera y lentamente el pie del embrague", pero nunca le dirá: "levante el pie 8° a una velocidad de 2°30' por segundo". Y sin embargo, el hombre, como sistema, se comporta bien (aprende) con entradas "borrosas" como las del primer tipo, mientras sería difícil que lo hiciera con las del segundo.

A poco que se reflexione,se llegará a la conclusión de que si se quieren analizar sistemas muy complejos como el hombre, las sociedades, etc. (ya sea con espíritu puramente científico, ya sea con fines utilitarios: construcción de mejores máquinas que puedan seguir procesos de decisión, que puedan - comprender el lenguaje natural, etc) resulta imprescindible introducir en los modelos la imprecisión o la subjetividad. Esta fue la idea que condujo a Zadeh, en 1.965, a definir los conjuntos borrosos o difusos ("fuzzy sets").

El concepto, además de nuevo, incide en la misma base de la matemática, por lo que choca un poco con la formación adquirida. Por otra parte, a la vista de la importancia que actualmente le dan los investigadores (basta consultar las actas de recientes Congresos sobre Matemáticas y Sistemas), parece - que sus aplicaciones en Ingeniería, y especialmente en Ordenadores (Inteligencia Artificial) pueden ser numerosas en el futuro. Estas consideraciones creemos que justifican tanto esta larga introducción como las páginas que siguen, en las que expondremos los principios básicos de la teoría introducida y -

desarrollada por Zadeh.

3.2. Subconjuntos borrosos.

En la teoría clásica, dado un elemento x de un universo U, y un subconjunto, $A \subset U$, hay dos posibilidades: $x \in A$ o $x \notin A$. Puede definirse una función característica de pertenencia, μ_A, tal que $\mu_A(x) = 1$ si $x \in A$ y $\mu_A(x) = 0$ si $x \notin A$. Es fácil demostrar que:

$$\mu_{\bar{A}}(x) = 1 - \mu_A(x)$$

$$\mu_{A \cap B}(x) = \mu_A(x) \cdot \mu_B(x)$$

$$\mu_{A \cup B}(x) = \mu_A(x) + \mu_B(x) \quad \text{(suma lógica)}$$

Definición 3.2.1. Dado un universo U, un subconjunto borroso, $\underset{\sim}{A}$, de U, es un conjunto de pares

$$\{(x|\mu_A(x)\}, \forall x \in U,$$

donde $\mu_A(x)$ es una función que toma sus valores en un conjunto M llamado conjunto de pertenencia.

Generalmente se toma $M = \{m|m \in [0,1]\}$; si se hace $M = \{0,1\}$, entonces $\underset{\sim}{A}$ se reduce a un subconjunto ordinario, de manera que la teoría clásica de conjuntos es un caso particular de la teoría de conjuntos borrosos.

Ejemplos.

a) Si $U = \{1,2,3,\ldots,10\}$, podemos definir el subconjunto "varios", $\underset{\sim}{V}$, como:

$$\underset{\sim}{V} = \{3|0,5;\ 4|0,8;\ 5|1;\ 6|1;\ 7|0,8;\ 8|0,5\}$$

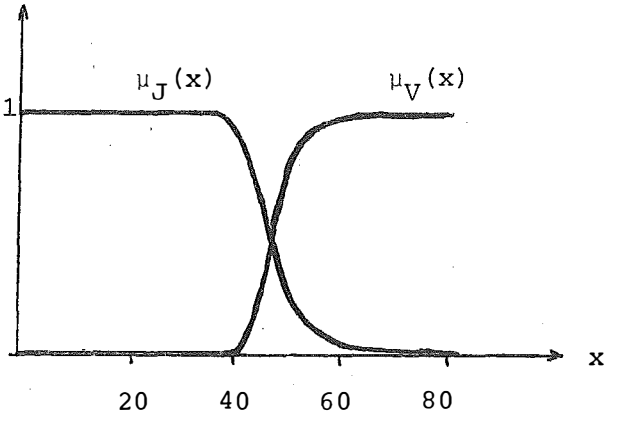
La asignación de valores a $\mu_V(x)$ es totalmente subjetiva, de manera que otra persona daría con toda probabilidad, otras cifras.

b) Si $U = \{x|x \in [\,0,150]\}$ se interpreta como el conjunto de edades posibles de un ser humano, podrían definirse los subconjuntos borrosos $\underset{\sim}{J}$ (joven) y $\underset{\sim}{V}$ (viejo) así:

$$\underset{\sim}{J} = \left\{(x|1)_{0 \leq x \leq 40};\ (x|(1+(x-40)^2/40)^{-1})_{x>40}\right\}$$

$$\underset{\sim}{V} = \left\{(x|0)_{0 \leq x \leq 40};\ (x|(1+40/(x-40)^2)^{-1}{}_{x>40}\right\}$$

Podemos tener una visión gráfica de estos conjuntos representando $\mu_J(x)$ y $\mu_V(x)$:



Definición 3.2.2. Definimos las relaciones de igualdad e inclusión y las operaciones de complementación, intersección y unión entre conjuntos borrosos del siguiente modo:

Igualdad: $\underset{\sim}{A} = \underset{\sim}{B}$ si $\mu_A(x) = \mu_B(x)$, $\forall x \in U$

Inclusión: $\underset{\sim}{A} \subset \underset{\sim}{B}$ si $\mu_A(x) \leq \mu_B(x)$, $\forall x \in U$

Complementación: $A = \bar{B}$ si $\mu_A(x) = 1-\mu_B(x)$, $\forall x \in U$ (supuesto que $M = [0,1]$)

Intersección: $\underset{\sim}{C} = \underset{\sim}{A} \cap \underset{\sim}{B}$ si $\mu_C(x) = \min(\mu_A(x), \mu_B(x))$, $\forall x \in U$

Unión: $\underset{\sim}{C} = \underset{\sim}{A} \cup \underset{\sim}{B}$ si $\mu_C(x) = \max(\mu_A(x), \mu_B(x))$, $\forall x \in U$

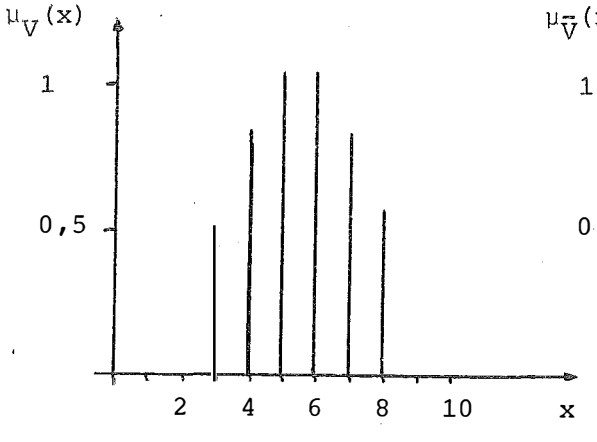
(Es fácil comprobar la equivalencia con las definiciones clásicas, para el caso de $M = \{0,1\}$).

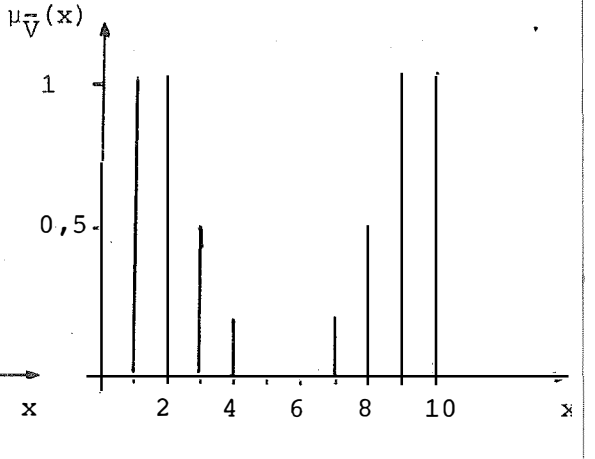
Ejemplos.

a) El complemento de "varios" tal como se definió más arriba sería:

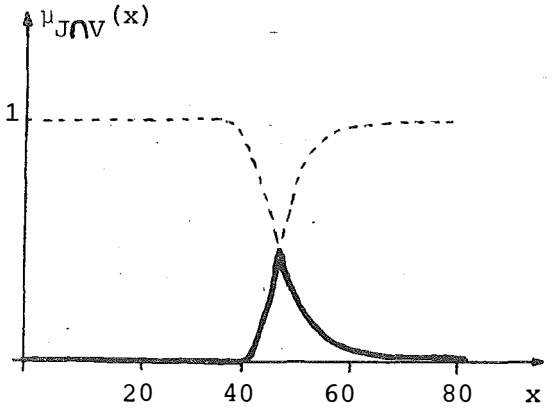
$$\underset{\sim}{\bar{V}} = \{1|1;\ 2|1;\ 3|0,5;\ 4|0,2;\ 7|0,2;\ 8|0,5;\ 9|1;\ 10|1\}$$

Gráficamente:

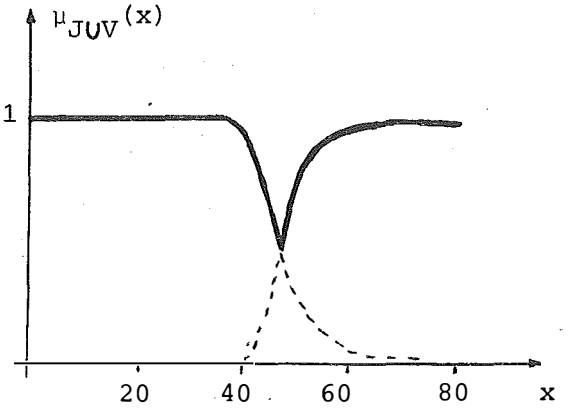




b) Con las definiciones anteriores de "joven" y "viejo" la intersección y la unión de ambos conjuntos se representarán gráficamente así:



"joven y viejo"



"joven o viejo"

Puede comprobarse que $\underset{\sim}{V} = \bar{\underset{\sim}{J}}$, viendo que $\mu_J(x)+\mu_V(x) = 1, \forall x$.

3.3. Lógica borrosa.

Si contemplamos a las variables lógicas binarias como funciones de pertenencia de conjuntos ordinarios, la lógica proposicional (y lo mismo el álgebra de conmutación) puede considerarse asociada a la teoría clásica de conjuntos. En efecto, dados los subconjuntos A y B de U, si llamamos $a = \mu_A(x)$ y $b = \mu_B(x)$ a sus funciones de pertenencia, entonces podemos definir entre a y b las operaciones

$$\bar{a} = 1-a = \mu_{\bar{A}}(x)$$

$$a.b = \mu_{A\cap B}(x)$$

$$a+b = \mu_{A\cup B}(x)$$

Se puede comprobar que <P, +, ., ->, donde P es el conjunto de las funciones de pertenencia de los subconjuntos de un universo U, es un álgebra de Boole. Las operaciones +,.,-, corresponden, respectivamente, a las conectivas $\vee$, $\wedge$, $\neg$ de la lógica proposicional.

Siguiendo un camino paralelo, sea un universo,U, en el que consideramos unos subconjuntos borrosos $\underset{\sim}{A} \subset U$, $\underset{\sim}{B} \subset U$. Para un elemento cualquiera $x \in U$ podemos definir unas variables borrosas, que serán los valores de las funciones de pertenencia de ese elemento a cada subconjunto:

$$\underset{\sim}{a} = \mu_A(x); \quad \underset{\sim}{b} = \mu_B(x) \qquad \underset{\sim}{a},\underset{\sim}{b} \in M = [0,1]$$

En el conjunto P de variables borrosas se definen las operaciones de complementación, producto y suma del siguiente modo:

$$\bar{\underset{\sim}{a}} = 1 - \underset{\sim}{a}$$

$$\underset{\sim}{a}.\underset{\sim}{b} = \min\ (\underset{\sim}{a},\underset{\sim}{b})$$

$$\underset{\sim}{a}+\underset{\sim}{b} = \text{máx}\ (\underset{\sim}{a},\underset{\sim}{b})$$

Con estas operaciones se pueden construir sentencias borrosas, de igual modo que en lógica proposicional clásica (Cap. 2, Ap. 1.2).

Puede comprobarse que se satisfacen todas las condiciones de un álgebra de Boole, salvo las que se refieren al elemento inverso ($a+\bar{a}=1$ y $a.\bar{a}=0$). En efecto:

$$\underset{\sim}{a} + \bar{\underset{\sim}{a}} = \underset{\sim}{a} + (1 - \underset{\sim}{a}) = \text{máx}\ (\underset{\sim}{a},\ 1 - \underset{\sim}{a})$$

sólo es 1 en dos casos: para $\underset{\sim}{a} = 0$, y para $\underset{\sim}{a} = 1$, pero, en general, $\underset{\sim}{a} + \bar{\underset{\sim}{a}} \neq 1$, y análogamente ocurre con

$$\underset{\sim}{a}.\bar{\underset{\sim}{a}} = \underset{\sim}{a}.(1 - \underset{\sim}{a}) = \min (\underset{\sim}{a}, 1-\underset{\sim}{a}) \neq 0 \quad \forall \underset{\sim}{a} \neq 0,1$$

Por ello, <P, +, ., -> no es un álgebra de Boole.

Se observará que las operaciones "-", "." y "+" definidas más arriba son idénticas a las "¬", "∧" y "∨" definidas por las leyes de Lukasiewicz. Sin embargo, la lógica borrosa no es idéntica a la lógica probabilista; se ha demostrado (*) que ambas son particularizaciones, con diferentes restricciones, de un tipo de lógica más general.

El uso de sentencias borrosas, junto con otros dos conceptos derivados, las variables lingüísticas y los algoritmos borrosos, son los tres pilares del enfoque propuesto por ZADEH (1.973) para el análisis de sistemas complejos y procesos de decisión.

Por otra parte, existen varios modos de realización electrónica de funciones de conmutación borrosas, y también se han ideado diversos métodos para su minimización.

(*) GAINES, B.R. Stochastic and fuzzy logics. Electronics letters, 11,9 (mayo 1.975), 188-189.

BIBLIOGRAFIA

1. OBRAS DE ORIENTACIÓN FILOSÓFICA Y MATEMÁTICA.

CHENIQUE, F. Comprendre la logique moderne. Dunod, París, - 1.974.

Volumen 1: Lógica proposicional, de clases y de predicados.

Volumen 2: Lógicas no clásicas.

DEAÑO,A. Introducción a la lógica formal. Alianza Universidad, Madrid, 1.975. (2^a ed., reimpr., 1.977).

Tiene características muy similares al libro de FERRÀTER; es algo más riguroso en cuanto al formalismo y también algo más profundo. Consta de dos volúmenes: el primero sobre lógica proposicional, y el segundo sobre lógica de predicados y algo de lógicas no clásicas. Contiene ejercicios de deducción muy interesantes.

FERRATER MORA, J. y LEBLANC, H. Lógica Matemática. Fondo de Cultura Económica, México, 1.955 (2^a ed. rev. 1.962).

Es una exposición no muy formalizada y de muy agradable lectura, que cubre desde la lógica proposicional - hasta los principios de las lógicas polivalentes.

GILBERT, W.J. Modern algebra with applications. Wiley, New York, 1.976.

En el Capítulo 2, sobre álgebra de Boole, incluye aplicaciones a la lógica y a los circuitos.

GLORIOSO, R.M. Engineering cybernetics. Prentice-Hall, Englewood Cliffs, n.J., 1.975.

Dedica un Capítulo (el 10) a la lógica de umbral.

PONASSE, D. Logique Mathématique. OCDL, París, 1.972. Versión americana: Mathematical logic, Gordon and Breach, New York, 1.973.

Extraordinariamente conciso y formalizado.

2. OBRAS ORIENTADAS AL DISEÑO DE CIRCUITOS DE CONMUTACIÓN.

MANDADO, E. Sistemas electrónicos digitales. Marcombo, Barcelona, 1.975 (3ª ed., ampliada, 1.977).

MUÑOZ, E. Circuitos electrónicos digitales II. Dpto. Publicaciones E.T.S.I.T.M., Madrid, 1.978.

NAGLE, H.T., CARROL, B.D. y IRWIN, J.D. An introduction to computer logic. Prentice Hall, Englewood Cliffs, N.J., 1.975.

3. PUBLICACIONES SOBRE LOS CONJUNTOS Y LA LÓGICA BORROSA.

KAUFMANN, A. Introduction a la théorie des sous-ensembles - flous a l'usage des ingénieurs. Masson, París, 1973-75.

Tomo 1: Elementos teóricos de base.
Tomo 2: Aplicaciones a la lingüística, a la lógica y a la semántica.
Tomo 3: Aplicaciones a la clasificación y al reconocimiento de formas, a los autómatas y a los sistemas, a la elección de criterios.

Es, hasta el presente, uno de los pocos libros (y probablemente el más completo) dedicados al tema. En las revistas y en las actas de los congresos pueden encontrarse numerosas aplicaciones; a título de ejemplo, citemos dos: una a sistemas de conmutación borrosos y su posible utilidad en control de calidad, procesos indus

triales y reconocimiento de formas, y la otra al análisis de sistemas complejos y procesos de decisión:

MARINOS, P.N. Fuzzy logic and its application to switching systems. IEEE Trans. Computers, C-18, (1.969), 343-348

ZADEH, L.A. Outline of a new approach to the analysis of complex systems and decision processes. IEEE Trans. Systems, Man, and Cybernetics, SMC-3 (1.973), 28-44.

TEMA 2

AUTOMATAS

GREGORIO FERNANDEZ FERNANDEZ

INDICE

Pág

Pág.

Pag.

Pag.

CAPITULO 1.

IDEAS GENERALES

1. AUTÓMATAS, ORDENADORES Y TRATAMIENTO DE LA INFORMACIÓN

La palabra "autómata", en el lenguaje ordinario, normalmente evoca algo que pretende imitar funciones propias de los seres vivos, especialmente las relacionadas con el movimiento. (Véase, para corroborar esta aserción, la definición de un diccionario cualquiera). En este sentido, un ejemplo de autómata sería el típico robot antropomorfo o zoomorfo dotado de capacidades autónomas de movimiento que le permiten ejecutar las órdenes o seguir el programa establecido por un ser inteligente. Así, el famoso personaje de Mary Shelley obedece al Dr. Frankestein "como un autómata".

En el campo de la Informática lo fundamental no es la simulación del movimiento, sino la simulación de los procesos de tratar la información (*), y el ejemplo típico de autómata no es ya el robot mecánico sino el ordenador (**).

(*) Esta idea la expuso ya Torres Quevedo (véase la explicación de la cubierta), antes de que existieran los ordenadores y la Informática.

(**) Por supuesto, el robot sigue siendo un autómata. Existe un campo de trabajo, llamado "robótica" o "robotología", que consiste en el diseño de robots, normalmente controlados por ordenador, para aplicaciones industriales o aeroespaciales o con el objetivo de estudiar, a través de la simulación, funciones propias de los seres vivos.

Pensemos en la naturaleza del trabajo que realizan los ordenadores. Los conocimientos básicos de que ya disponemos nos permiten llegar a la conclusión de que un ordenador no es más que un dispositivo que manipula símbolos. Un ejemplo será útil para reforzar esta idea:

Consideremos un ordenador con palabras de 16 bits. Supongamos que este ordenador recibe de un periférico una serie de impulsos que se graban en una determinada posición de su memoria dejando en ella la siguiente configuración de bits:

1010010011000001

¿Qué significa esto para el ordenador?. Entre otras muchas cosas puede ser:

- el número -23359 expresado en binario con convenio de complemento a 2.
- Los caracteres $A codificados en código ASCII, con el bit 8 puesto siempre a 1 (convenio seguido en algunos ordenadores).
- Una instrucción del lenguaje de máquina del ordenador.

El que sea una u otra cosa depende de dos personas:

- el diseñador del ordenador (para ser realista habría que hablar del equipo de diseño), que decidió que los números se representen en binario y complemento a 2, o que, interpretado como instrucción, ese código de operación signifique, por ejemplo, "sumar", y no otra cosa, etc.;
- el utilizador, que, al hacer su programa, decide que en un momento dado el ordenador lleve esa configura-

ción de bits de la memoria al acumulador, o al registro de instrucción, o a la unidad de salida, etc. Al tomar tal decisión, el utilizador está dotando al conjunto de bits de una significación y, por consiguiente, de una capacidad para representar información. Para el ordenador, sin embargo, los bits no son más que símbolos materializados por el sentido de imantación de unos núcleos de ferrita, o por los niveles de tensión de determinadas puertas.

Por consiguiente, cuando se dice que el ordenador es un "equipo" capaz de realizar automáticamente tratamientos (operaciones aritméticas, comparaciones, etc.) de información (númerica, alfanumérica, etc.) (ver KUBUSCH y CARDENAS, 1.977, pág. 190) debe entenderse que este tratamiento o procesamiento de la información sólo tiene sentido para nosotros, que decidimos que tal "tira" o "cadena" de símbolos significa tal cosa, es decir, que *codificamos la información en cadenas de símbolos*; el ordenador se limita a manipular esas cadenas, dando normalmente como resultado otras cadenas que nosotros decodificamos.

Y hablando ya en términos generales, podemos considerar a un autómata como un dispositivo que manipula cadenas de símbolos que se le presentan a su entrada, produciendo otras tiras o cadenas de símbolos como salida. En el Apartado 3.1.- del Capítulo 2 formalizaremos matemáticamente esta definición.

Un ordenador es un ejemplo de autómata, pero también son autómatas dispositivos más sencillos,como sumadores, contadores, etc., que veremos como ejemplos en el Capítulo 2, o las mismas partes constituyentes de un ordenador: la unidad aritmética-lógica o la unidad de control son autómatas. Por otra parte, existen autómatas más complejos que un ordenador, como los robots que mencionábamos antes en nota a pie de página. Tam-

bién pueden estudiarse como autómatas determinadas funciones - de los seres vivos, e incluso complejos sistemas ecológicos y socioeconómicos.

2. Autómatas y máquinas secuenciales. Concepto de estado.

El autómata recibe los símbolos de entrada uno detrás de otro, distribuidos en el tiempo, es decir, secuencialmente. Además, en general, el símbolo de salida que en un instante determinado produce este autómata no sólo depende del último símbolo recibido a la entrada, sino de toda la secuencia o cadena, distribuida en el tiempo, que ha recibido hasta ese instante.- Quiere esto decir que un autómata es una máquina secuencial, - en el sentido de que opera sobre secuencias de símbolos, "re-- cordando" en todo instante la "historia" de símbolos llegados hasta ese instante. Esto le diferencia de una máquina puramente combinatoria como sería, por ejemplo, un circuito lógico de los estudiados en el Tema "Lógica".

Así pues, ante un determinado símbolo de entrada un - autómata puede producir diferentes símbolos de salida, depen-- diendo de la historia o secuencia de todos los símbolos de entrada anteriores.

Obsérvese que hasta ahora venimos hablando de un autómata como una "caja negra" con una entrada en la que recibe - símbolos y una salida, sobre la que deposita otros símbolos. - Otra manera de enfocar el estudio de los autómatas es considerando lo que hay dentro de la caja negra (aunque sea de una manera abstracta, es decir, prescindiendo de la naturaleza de - los componentes físicos y atendiendo sólo a las transformaciones de símbolos), y esto nos lleva a definir un concepto funda

mental: el estado del autómata. *El estado es toda la información necesaria en un momento dado para poder deducir, dado un símbolo de entrada en ese momento, cual será el símbolo de salida*. Es decir, conocer el estado es lo mismo que conocer toda la historia de símbolos de entrada (*). Un autómata tendrá un determinado número de estados (en teoría, puede tener infinitos), y se encontrará en uno u otro según sea la historia de símbolos que le han llegado; si encontrándose en un estado determinado, recibe un símbolo también determinado, producirá un símbolo de salida y efectuará un cambio o *transición* a otro estado (también puede quedarse en el mismo). Estas ideas son fáciles de formalizar matemáticamente a partir de los conceptos de conjunto y función, y conducen a la definición de autómata que desarrollaremos en el Capítulo siguiente.

3. Autómatas y lenguajes

Un campo importante dentro de la Informática, al que dedicaremos el último Tema, está constituído por el estudio de los lenguajes y las gramáticas que los generan. Los elementos de un lenguaje son sentencias, palabras, etc., formados a partir de un *alfabeto*, que no es otra cosa que un conjunto finito de símbolos. Establecidas unas reglas gramaticales, una cadena de símbolos pertenecerá al correspondiente lenguaje si tal cadena se ha formado obedeciendo esas reglas; puede entonces pensarse en la posibilidad de construir un *autómata reco*

(*) Realmente, no basta con conocer toda la historia de símbolos de entrada para saber cuál es la salida; es necesario conocer también el "estado inicial", es decir, el estado en que se encontraba el autómata al recibir el primero de los símbolos de entrada.

cedor de ese lenguaje, tal que cuando reciba a su entrada una determinada secuencia de símbolos produzca, por ejemplo, un - "1" a la salida si la secuencia es correcta, y un "0" si no - lo es. De este modo, como veremos en su momento, a cada tipo de gramática corresponde un tipo de autómata.

4. AUTÓMATAS Y ÁLGEBRA

Las cadenas de entrada y salida de un autómata se forman a partir de los correspondientes alfabetos mediante una - operación que consiste en poner los símbolos unos a continuación de otros. Esta operación se llama concatenación, y es - asociativa. Por consiguiente, el conjunto de todas las cadenas con la concatenación tiene una estructura algebráica de - semigrupo; si, además, definimos un elemento neutro, tendremos un monoide.

Por otra parte, como veremos en el Capítulo siguiente, si el autómata es finito (es decir, si tiene un número finito de estados) puede determinarse un número finito de clases de equivalencia en el semigrupo (o monoide) de entrada, lo cual permite definir un semigrupo (o monoide) cociente llamado el semigrupo (o monoide) de la máquina, a partir del cual pueden formalizarse muchas cuestiones relativas al funcionamiento de los autómatas.

Siguiendo esta línea de trabajo, se ha elaborado en - las dos últimas décadas una teoría abstracta de autómatas con una fuerte base algebráica que, según ARBIB (1.969), constituye "la matemática pura de la Informática".

5. Resumen y conclusiones

a) La Teoría de Autómatas, también llamada Teoría algebráica de máquinas, permite estudiar de un modo sistemático las máquinas, más o menos complicadas, que realizan un procesamiento de la información y que actúan de manera discreta, es decir, la información se supone codificada a partir de un conjunto finito de símbolos que el autómata trata secuencialmente, uno detrás de otro. La Teoría de Autómatas proporciona métodos para el análisis y la síntesis de tales máquinas.

b) Los trabajos sobre lenguajes y gramáticas formales han evolucionado en una dirección que les ha conducido a encontrarse con la Teoría de Autómatas como herramienta matemática de gran utilidad.

c) La Teoría de Autómatas no sólo puede aplicarse a las "máquinas", en el sentido estricto que normalmente damos a esta palabra, sino también a muchos sistemas naturales, y, en general, permite estudiar procesos que dependen de una "historia", es decir, cuyo comportamiento presente es función del pasado.

d) En sus veinte años de historia la Teoría de Autómatas se ha constituído en una disciplina muy formalizada que sigue en evolución. Mientras la teoría básica (autómatas finitos deterministas) puede considerarse definitivamente establecida, se abren nuevas vías que actualmente son objeto de estudio de los investigadores y que ofrecen amplias perspectivas de aplicación: autómatas estocásticos, borrosos, adaptativos, de aprendizaje, etc.

e) Evidentemente, en este Tema no podemos exponer ni siquiera resumir, toda la Teoría de Autómatas. Nuestro objeti-

vo será presentar los principios básicos, desarrollando algunos ejemplos de aplicación para ver su utilidad práctica en - diversos campos, especialmente el de la Informática.

CAPITULO 2.

AUTOMATAS FINITOS

1. DEFINICIÓN Y REPRESENTACIÓN DE LOS AUTÓMATAS

1.1. Definición

Un autómata es una quíntupla:

$$A = \langle E, S, Q, f, g \rangle \ , \qquad [\ 1.1.1\ ]$$

donde:

E es un conjunto finito, llamado conjunto de entradas o alfabeto de entrada, cuyos elementos llamaremos entradas o símbolos de entrada.

S es un conjunto finito, llamado conjunto de salidas o alfabeto de salida, cuyos elementos llamaremos salidas o símbolos de salida.

Q es un conjunto llamado conjunto de estados.

f es una función $f: E \times Q \to Q$, llamada función de transición o función de estado siguiente.

g es una función $g: E \times Q \to S$, llamada función de salida.

Esta definición formal puede interpretarse como la descripción matemática de una máquina que, si en el instante

t recibe una entrada e ∈ E y se encuentra en el estado q ∈ Q, entonces da una salida g (e,q), y pasa al estado f (e,q) en - el instante t + 1. (Suponemos una escala discreta de tiempos arbitraria: t = 1,2,3....). Expresando de una manera explícita el tiempo, y si llamamos s a un elemento genérico de S, podemos escribir:

$$q\ (t + 1) = f[\ e(t),\ q(t)\ ]\ ;\ s(t) = g\ [\ e(t),\ q(t)\ ]$$

A es un *autómata finito* si Q es un conjunto finito. En lo sucesivo (en este Tema) hablaremos casi siempre de autómatas finitos, y abreviaremos escribiendo "AF".

1.2. Representación

1.2.1. Tabla de transiciones

Las funciones f y g pueden representarse mediante una tabla con tantas filas como estados y tantas columnas como entradas. Si la fila i corresponde al estado q_i y la columna j corresponde a la entrada e_j, en la intersección de ambas se escribirá f (e_j,q_i)/g (e_j,q_i). Por ejemplo, sea el AF definido - por los conjuntos

$$E = \{a,b\}$$

$$S = \{0,1\}$$

$$Q = \{q_1,q_2,q_3\}$$

y las funciones de estado y de salida

$$f(a,q_1) = q_1; \quad g(a,q_1) = 0$$

$$f(b,q_1) = q_2; \quad g(b,q_1) = 1$$

$$f(a,q_2) = q_3; \quad g(a,q_2) = 0$$

$$f(b,q_2) = q_2; \quad g(b,q_2) = 0$$

$$f(a,q_3) = q_3; \quad g(a,q_3) = 1$$

$$f(b,q_3) = q_1; \quad g(b,q_3) = 0$$

En lugar de esto, es más cómodo representar f y g por la tabla de transiciones de la figura 2.1.

e / q	a	b
q_1	$q_1/0$	$q_2/1$
q_2	$q_3/0$	$q_2/0$
q_3	$q_3/1$	$q_1/0$

Fig. 2.1.

1.2.2. Diagrama de Moore

Otra forma de representar las funciones f y g es mediante un grafo orientado en el que cada nodo corresponde a un estado, y si $f(e,q_i) = q_j$ y $g(e,q_i) = s$, existe un arco dirigido del nodo correspondiente a q_i al correspondiente a q_j, sobre el que pondremos la etiqueta e/s. Por ejemplo, el AF defi-

nido por la tabla anterior puede representarse por el grafo de la Fig. 2.2. Este grafo suele llamarse diagrama de transiciones o diagrama de Moore.

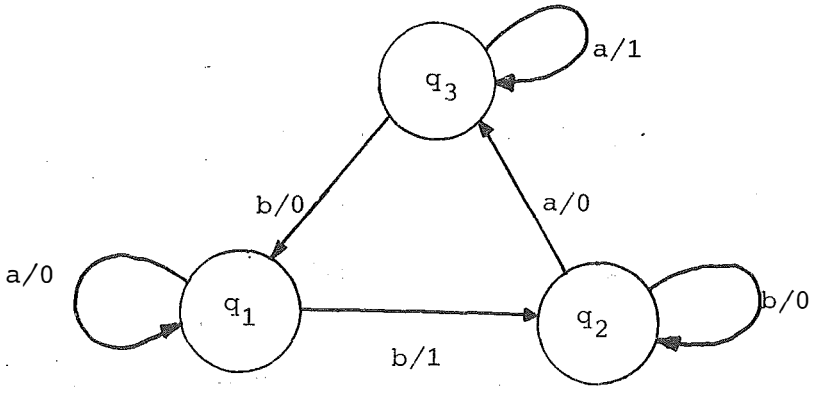


Fig. 2.2.

1.3. Máquinas de Moore y de Mealy

El modelo general de autómata que hemos definido se llama *máquina de Mealy*. Las funciones f y g determinan la salida y el estado siguiente cuando la máquina se encuentra en un estado $q \in Q$ y recibe una entrada $e \in E$. Ahora bien, por conveniencia matemática, es interesante considerar, además de los símbolos o elementos de E, un elemento neutro, Λ; físicamente, el decir que la entrada es Λ es lo mismo que decir que no hay ninguna entrada. Es inmediato entonces plantearse la siguiente pregunta: ¿qué ocurre si, estando un autómata en el estado $q \in Q$, recibe como entrada Λ?. Para responder a esto, matemáticamente, habría que ampliar el dominio de f, que es $E \times Q$, a $\{E \cup \{\Lambda\}\} \times Q$, y lo mismo el dominio de g. La ampliación del dominio de f no plantea ningún problema: se puede convenir que $f(\Lambda,q) = q$ (es decir, físicamente, que si no hay entrada no se cambia de estado). Pero no ocurre lo mismo con g; y ello se ve fácilmente si nos referimos al ejemplo desarrollado más arriba (Fig.2.1): si llegamos a q_1 ya sea de q_3 (por efecto de entrada b) o de q_1 (por a) la salida es 0, por lo que podemos asociar la salida 0 al estado q_1 y decir $g(\Lambda,q_1) = 0$; sin embargo, no

podemos definir g (Λ,q_2), ya que si llegamos a q_2 desde q_1 la salida es 1, mientras que si llegamos desde el propio q_2 la salida es 0. Es evidente que, en general, sólo puede definirse g (Λ,q) en el caso en que se cumpla que

$$[q = f(e_1,q_1) = f(e_2,q_2)] \rightarrow [g(e_1,q_1) = g(e_2,q_2)] \qquad [\ 1.3.1\ ]$$

es decir, que a q se le pueda asociar una salida y una sola. Si esto ocurre para todo $q \in Q$ podemos definir una función inyectiva $h: Q \rightarrow S$ tal que $g(e,q) = h[f(e,q)]$, $e \in \{E \cup \{\Lambda\}\}$, $q \in Q$. En este caso, podemos decir que la salida sólo depende del estado, y el autómata se llama *máquina de Moore*. Expresando el tiempo de manera explícita:

$$s(t) = g[e(t), q(t)] = h[q(t)] = h[f[e(t-1),q(t-1)]]$$

En una máquina de Mealy las salidas están asociadas con las transiciones, mientras que en una máquina de Moore las salidas están asociadas a los estados, o, lo que es lo mismo, todas las transiciones que conducen a un mismo estado tienen asociada la misma salida. También podemos decir que una máquina de Mealy, en el instante de efectuar una transición necesita conocer una entrada $e \in E$ (ya que, en general, g (Λ,q) no está definida), mientras que en una máquina de Moore la entrada puede ser $e \in E$ o $e = \Lambda$.

Puesto que toda máquina de Moore es una máquina de Mealy que cumple la condición [1.3.1] para todo $q \in Q$, parece en principio que las primeras son un subconjunto de las segundas. Sin embargo, vamos a demostrar que, dada una máquina de Mealy, siempre podremos encontrar una máquina de Moore equivalente (normalmente, a costa de aumentar el número de estados). En efecto, si tenemos una máquina de Mealy

$$A = \langle E, S, Q, f, g \rangle ,$$

siempre podemos definir un nuevo autómata

$$\hat{A} = \langle E, S, \hat{Q}, \hat{f}, \hat{g} \rangle,$$

en el que $\hat{Q}$ se obtiene escindiendo cada $q \in Q$ en tantos estados q^s (*) como salidas s puedan asociarse a q:

$$\hat{Q} = \{q^s \mid \exists (q' \in Q \text{ y } e \in E) \text{ tales que } f(e,q') = q, \text{ y } g(e,q') = s\}$$

y en el que $\hat{f}$ y $\hat{g}$ se definen así:

$$\hat{f}(e,q^s) = [f(e,q)]^{g(e,q)}$$

$$\hat{g}(e,q^s) = g(e,q)$$

De este modo, a cada $q^s \in \hat{Q}$ se le puede asociar una sola salida, s, y así tendremos una función de salida actual $\hat{h}: \hat{Q} \to S$ - tal que $\hat{g}(e,q^s) = \hat{h}[\hat{f}(e,q^s)]$, por lo que $\hat{A}$ será una máquina de Moore.

Concretemos estas ideas con un ejemplo. Tomemos el - - autómata cuyo diagrama es el de la figura 2.2. Como ya hemos - visto, con q_1 siempre se puede asociar la salida 0; sin embargo, q_2 lo escindiremos en q_2^0 y q_2^1, ya que la salida asociada - es 0 ó 1 según que vengamos de q_2 o de q_1, y, del mismo modo, escindiremos q_3 en q_3^0 y q_3^1. De acuerdo con esto, y teniendo en cuenta las definiciones de f y g obtenemos el diagrama de la - Fig. 2.3.

(*) s aquí es un superíndice, no un exponente.

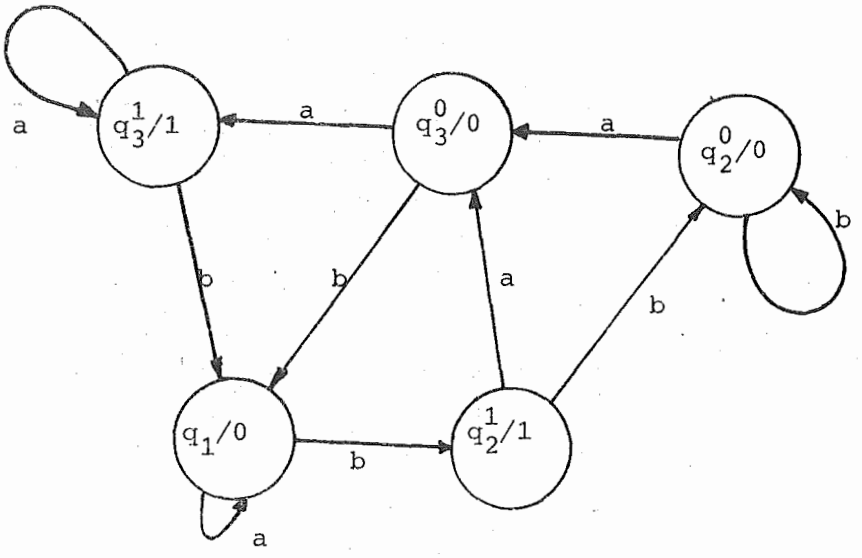


Fig. 2.3.

Obsérvese que, al estar las salidas asociadas con los estados, todas las transiciones que conducen a un estado producen la misma salida, por lo que en lugar de rotular las salidas sobre los arcos las hemos incluido en los nodos. Del mismo modo, en la tabla de transiciones podemos incluir las salidas en la misma columna de estados; la tabla de esta máquina de Moore es entonces la de la figura 2.4.

e / q/s	a	b
$q_1/0$	q_1	q_2^1
$q_2^0/0$	q_3^0	q_2^0
$q_2^1/1$	q_3^0	q_2^0
$q_3^0/0$	q_3^1	q_1
$q_3^1/1$	q_3^1	q_1

Fig. 2.4

En lo sucesivo siempre que hablemos de un autómata supondremos, a menos que se diga lo contrario, que se trata de una máquina de Moore, es decir, representaremos indistintamente la salida por la función de salida g(e,q) o por la función de salida h(q) teniendo en cuenta que g = h∘f.

1.4. Ejemplos de aplicación

1.4.1. Detector de paridad

Un procedimiento sencillo y muy utilizado para detectar errores en una transmisión digital (por ejemplo, en una transferencia de datos de un periférico remoto a la memoria central) consiste en enviar un bit de paridad. Este bit puede ser tal que haga par el número total de "unos" enviados (paridad par), o que lo haga impar (paridad impar). Por ejemplo, supongamos que el periférico envía caracteres codificados en código ASCII de 8 bits con paridad par (es decir, el código es ASCII de 7 bits, y el octavo bit es el de paridad). El carácter A, en ASCII de 7 bits, se codifica 1000001; luego en 8 bits será 01000001 (el bit de paridad se hace 0 para que el número total de "unos" sea par). Por el contrario, el código de "C" es 1000011, por lo que el bit de paridad deberá ser 1 y por consiguiente en 8 bits será 11000011.

En el punto emisor deberá existir un *generador de paridad*, y en el receptor, un *detector de paridad*. Este detector deberá dar una señal de error en el caso de que la paridad recibida no sea correcta, cosa que ocurrirá cuando en la transmisión haya habido una alteración en un bit (o en un número impar de bits). Si la paridad es correcta no dará error. (Obsérvese que si hay un número par de alteraciones en la transmisión este sistema no detecta el error, pero la probabilidad de que ocurra más de una alteración es muy pequeña. Existen otros pro

cedimientos más complicados de detección e incluso corrección de errores que se estudian en otras asignaturas, principalmente en "Transmisión digital").

El alfabeto de entrada del detector es, evidentemente, E = {0,1}. El alfabeto de salida constará de dos elementos ("error" y "no error"); podemos tomar el convenio de que sea también S = {0,1}, donde "0" significa "no error" y "1" significa "error". El conjunto de estados puede ser Q = $\{q_0, q_1, q_2\}$, donde q_0 es el estado inicial, del que sólo se sale al recibir el primer bit; en q_1 se estará si se ha recibido un número par de bits y en q_2 si se ha recibido un número impar, de manera que, al finalizar la transmisión, si la máquina se ha quedado en q_1 es que no ha habido error (s=0), y si se ha quedado en q_2 es que sí lo ha habido (s=1). De acuerdo con esto, es fácil establecer el diagrama de Moore de la figura 2.5.

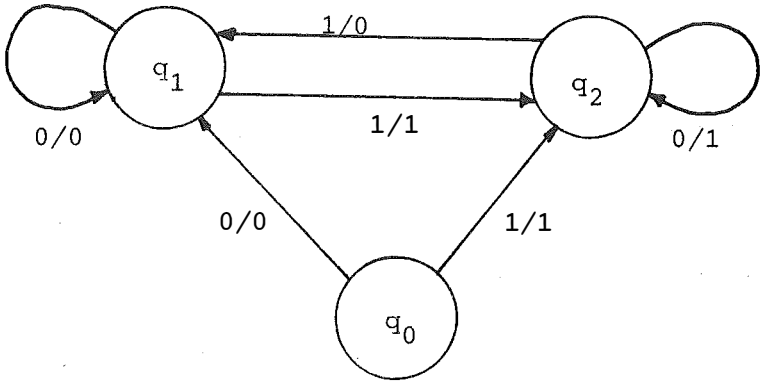


Fig. 2.5.

Ahora bien, el estado q_0 puede fundirse con el q_1. Ello equivale a convenir en que inicialmente, cuando no se ha recibido ningún bit (es decir, cuando se ha recibido Λ) la salida es 0. Obtenemos así el diagrama de Moore de la figura 2.6, en el que cada estado tiene una salida, y sólo una, aso-

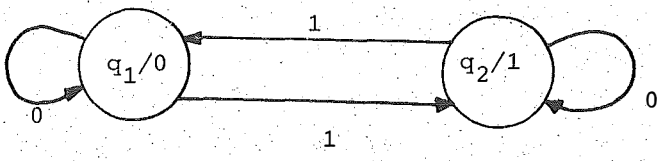


Fig. 2.6.

ciada (es una máquina de Moore).

1.4.2. Sumador binario serie

Un sumador binario es un dispositivo que suma dos números codificados en forma binaria y da el resultado también en binario. En el sumador serie los bits de los sumandos se presentan secuencialmente y por parejas, es decir, primero se presentan los dos bits de menor peso, el sumador los suma y obtiene el bit de menor peso del resultado (y toma nota del arrastre, si lo hay), luego los siguientes, etc. (Fig. 2.7).

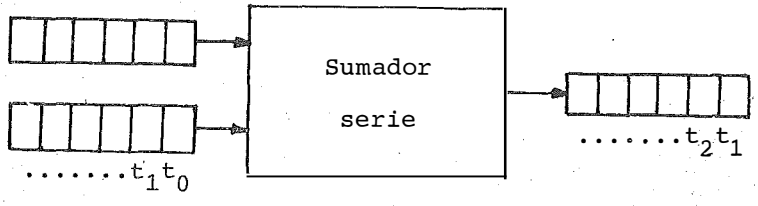


Fig. 2.7.

Evidentemente, el sumador serie es un autómata, puesto que, en todo momento, debe recordar si ha habido arrastre de los bits sumados anteriormente, es decir, la salida no sólo depende de la entrada actual, sino también de las anteriores. También es fácil ver que sólo necesita dos estados. En

efecto, en cada momento, para efectuar una suma de dos bits, sólo hay dos posibles situaciones a considerar: que no exista arrastre de los anteriores o que sí lo haya; llamemos q_1 y q_2, respectivamente, a los estados correspondientes a esas situaciones. Tenemos, por tanto:

$$E = \{00, 01, 10, 11\}$$

$$S = \{0, 1\}$$

$$Q = \{q_1, q_2\}$$

Inicialmente, el autómata estará en el estado q_1 (al recibir la primera pareja de bits no tiene que considerar ningún arrastre anterior). Si la primera pareja es 00, la salida deberá ser 0, y, como no hay arrastre, se quedará en q_1; si es 01 ó 10 deberá dar salida 1 y también quedarse en q_1, pero si recibe 11 la salida deberá ser 0, y habrá arrastre, por lo que pasará a q_2. Estando en q_2, si recibe 00, como hay arrastre de la suma anterior, deberá dar como salida 1 pero ya no habrá arrastre para la suma siguiente, por lo que pasará a q_1; sin embargo, en cualquier otro caso (01, 10, 11) se quedará en q_2, ya que sigue existiendo arrastre. Toda esta descripción se puede expresar con mayor concisión y claridad con el diagrama de Moore de la figura 2.8.

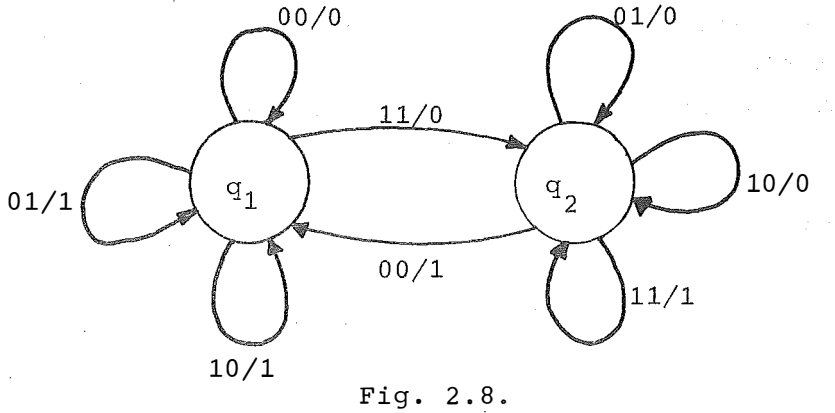


Fig. 2.8.

Este AF es una máquina de Mealy, puesto que tanto q_1 - como q_2 tienen asociadas las salidas 0 y 1. La máquina de Moore equivalente puede encontrarse siguiendo el procedimiento expuesto en el Apartado 1.3, y resulta ser la descrita por el - diagrama de la figura 2.9.

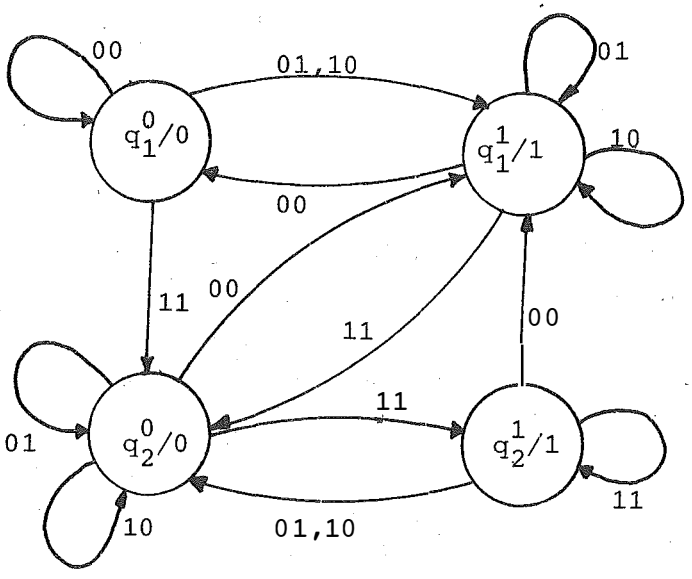


Fig. 2.9.

1.4.3. El castillo encantado

El siguiente ejemplo, tomado de ASHBY (1.956), nos - servirá para ilustrar cómo la Teoría de Autómatas tiene un - campo de aplicación muy extenso: todo lo que se refiera a sistemas (en el más amplio sentido de la palabra) discretos con memoria; en este caso particular veremos cómo permite formalizar un problema de lógica pura que, por su naturaleza secuencial, sería difícil de plantear con los métodos de la lógica formal.

El problema es el expuesto en esta carta:

"Querido amigo: Al poco tiempo de comprar esta vieja

mansión tuve la desagradable sorpresa de comprobar que está - hechizada con dos sonidos de ultratumba que la hacen práctica mente inhabitable: un canto picaresco y una risa sardónica.

Aún conservo, sin embargo, cierta esperanza, pues la experiencia me ha demostrado que su comportamiento obedece a ciertas leyes, oscuras pero infalibles, y que puede modificar se tocando el órgano y quemando incienso.

En cada minuto, cada sonido está presente o ausente. Lo que cada uno de ellos hará en el minuto siguiente depende de lo que pasa en el minuto actual, de la siguiente manera:

El canto conservará el mismo estado (presente o ausen te) salvo si durante el minuto actual no se oye la risa y to- co el órgano, en cuyo caso el canto toma el estado opuesto.

En cuanto a la risa, si no quemo incienso, se oirá o no según que el canto esté presente o ausente (de modo que la risa imita al canto con un minuto de retardo). Ahora bien, si quemo incienso la risa hará justamente lo contrario de lo que hacía el canto.

En el momento en que le escribo estoy oyendo a la vez la risa y el canto. Le quedaré muy agradecido si me dice qué manipulaciones de órgano e incienso debo seguir para restable cer definitivamente la calma".

La carta, especialmente en su tercer párrafo, descri- be un sistema lógico secuencial que puede formalizarse como - un autómata finito. Hay dos variables de entrada (órgano e in cienso), y como cada una de ellas tiene dos valores posibles, tendremos cuatro entradas diferentes; llamémoslas e_0, e_1, e_2, e_3.

e_0: no tocar el órgano ni quemar incienso;

e_1: no tocar el órgano pero quemar incienso;

e_2: tocar el órgano pero no quemar incienso;

e_3: tocar el órgano y quemar incienso.

También son cuatro los estados posibles:

q_0: ni risa ni canto

q_1: no risa, sí canto

q_2: sí risa, no canto

q_3: risa y canto

En cuanto a la salida, podemos considerar dos situaciones:

1 : que haya algún sonido (salida asociada a los estados q_1, q_2, q_3)

0 : que no haya ningún sonido (salida asociada al estado q_0).

Con esta nomenclatura, el problema se puede expresar diciendo que nos encontramos en un estado inicial, el q_3, queremos pasar a un estado final, el q_0, y se trata de encontrar la secuencia de entrada adecuada.

Siguiendo el enunciado, podemos obtener la tabla y el diagrama de transiciones, pero ello resulta mucho más fácil si utilizamos un formalismo lógico. Designemos por I y O unas variables booleanas que representen el incienso y el órgano, respectivamente (es decir, I=0 si no se quema incienso, I=1 -

si se quema, etc.), y por R y C otras variables que representan la risa y el canto (R=0 si no se oye la risa, etc.). Tenemos una correspondencia inmediata entre los valores de estas variables y los conjuntos de entradas y estados definidos más arriba:

e	O I	q	R C
e_0	0 0	q_0	0 0
e_1	0 1	q_1	0 1
e_2	1 0	q_2	1 0
e_3	1 1	q_3	1 1

Si en el minuto t los valores de estas variables son O_t, J_t, C_t, R_t, en el minuto t+1 tomarán los valores dados por las siguientes expresiones lógicas, que no son más que otra forma de expresar los párrafos 4 y 5 de la carta:

$$O_t . \bar{R}_t = 1 \longrightarrow C_{t+1} = \bar{C}_t \qquad (1)$$

$$O_t . \bar{R}_t = 0 \longrightarrow C_{t+1} = C_t \qquad (2)$$

$$I_t = 0 \longrightarrow R_{t+1} = C_t \qquad (3)$$

$$I_t = 1 \longrightarrow R_{t+1} = \bar{C}_t \qquad (4)$$

De (1) y (2) se deduce que

$$C_{t+1} = (O_t . \bar{R}_t) \oplus C_t \qquad (5)$$

y de (3) y (4)

$$R_{t+1} = I_t \oplus C_t \qquad (6)$$

De (5) y (6) se pueden sacar inmediatamente las tablas de verdad de C_{t+1} y R_{t+1} en función de O_t, I_t, C_t, R_t:

O_t	I_t	R_t	C_t	R_{t+1}	C_{t+1}
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	1	1	1
0	1	0	0	1	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	1
1	0	0	0	0	1
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	0	0
1	1	1	0	1	0
1	1	1	1	0	1

Volviendo ahora a la correspondencia establecida entre O, I, y e, y entre C,R y q, la anterior tabla de verdad nos conduce a la tabla de transiciones de la figura 2.10.

q/s \ e	e_0	e_1	e_2	e_3
$q_0/0$	q_0	q_2	q_1	q_3
$q_1/1$	q_3	q_1	q_2	q_0
$q_2/1$	q_0	q_2	q_0	q_2
$q_3/1$	q_3	q_1	q_3	q_1

Fig. 2.10.

y de aquí podemos dibujar el diagrama de Moore de la figura - 2.11

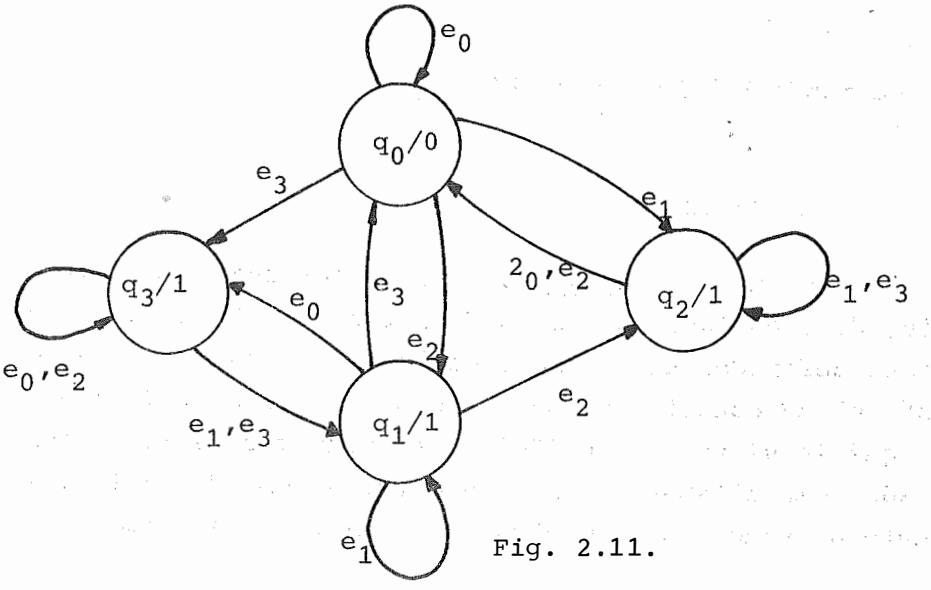


Fig. 2.11.

A la vista de este diagrama, la solución al problema planteado en la carta aparece fácilmente: pasar primero de q_3 a q_1 mediante e_1 o e_3, y luego a q_0 mediante e_3; o bien, respon

diendo en los mismos términos epistolares, "durante un minuto, queme usted incienso (tocando o no el órgano, es indiferente), y desaparecerá la risa; durante el minuto siguiente, queme incienso y toque el órgano, y al finalizar ese minuto cese toda actividad, con lo que, si no vuelve a manipular ni el órgano - ni el incienso, se habrá librado para siempre de tan molestos moradores".

Hay desde luego, otras soluciones, pero todas ellas - con secuencias de entrada más largas que la propuesta; por - - ejemplo, de q_1 puede pasarse a q_2 con e_2, y de aquí a q_0 con - e_0 ó e_2. Obsérvese que, afortunadamente para el propietario de la casa, el estado q_0 es estable, en el sentido de que con la entrada e_0 (es decir, I=0, O=0) el siguiente estado es el mismo q_0.

2. El monoide libre de entrada

2.1. Alfabeto

Llamaremos alfabeto a cualquier conjunto finito, no vacío, de símbolos. Ya sabemos (Apartado 1.1) que todo autómata tiene definidos un alfabeto de entrada y un alfabeto de salida. Así, el AF constituído por un ordenador conectado a un teletipo que le sirve como unidad de entrada y de salida tiene asociado como alfabeto, tanto de entrada como de salida, el conjunto de todos los símbolos que admite el teletipo.

2.2. Cadenas

Cadena (o palabra, o expresión, o secuencia finita) es

toda secuencia ordenada, finita, con o sin repetición, de símbolos de un alfabeto.

Obsérvese que los elementos o símbolos de un alfabeto pueden ser cadenas, no símbolos, en otro alfabeto. Así, en el ejemplo de 1.4.1 el alfabeto de entrada era {0,1}, mientras que en 1.4.2 era {00, 01, 10, 11}; evidentemente, los símbolos de este segundo alfabeto son cadenas para el primero.

Cadena vacía, Λ, es la que no contiene ningún elemento.

Longitud o grado de una cadena x, lg(x), es el número de símbolos de que consta x. En el caso de la cadena vacía, $lg(\Lambda) = 0$

2.3. El lenguaje universal y los lenguajes sobre un alfabeto.

Si E es un alfabeto, designemos por E^n el conjunto de todas las cadenas de longitud n. Por ejemplo, si E = {a,b}, $E^0 = \{\Lambda\}$, $E^1 = \{a,b\}$, $E^2 = \{aa, ab, ba, bb\}$, etc.

Definimos el lenguaje universal o universo de entradas de un alfabeto E como el conjunto infinito

$$E^* = E^0 \cup E^1 \cup E^2 \cup \ldots\ldots = \bigcup_{i=0}^{\infty} E^i$$

Es decir, E^* es el conjunto de todas las cadenas (incluída Λ) que pueden formarse a partir de E.

Todo subconjunto de E^* se llama lenguaje sobre E

2.4. Concatenación

La concatenación o yuxtaposición de dos cadenas, x, $y \in E^*$, es una ley de composición interna sobre E^*, es decir, una aplicación $E^* \times E^* \to E^*$, que consiste en formar la cadena xy poniendo y a continuación de x.

Por ejemplo, si A = {a,b,c} y tenemos las cadenas x = ab; y = bba, entonces xy = abbba.

La concatenación de una cadena consigo misma puede abreviarse mediante notación exponencial: $x^0 = \Lambda$, $x^1 = x$, $x^2 = xx$; $x^3 = xxx$, etc.

Obsérvese que la longitud cumple respecto a la concatenación las mismas propiedades formales que el logaritmo respecto a la multiplicación:

$$lg(xy) = lg(x) + lg(y)$$

$$lg(x^n) = n lg\ (x)$$

2.5. Monoide generado por un alfabeto

La operación de concatenación tiene las siguientes propiedades:

a) Es asociativa, es decir, x(yz) = (xy)z = xyz

b) En general, no es conmutativa: $xy \neq yx$

d) Tiene un elemento neutro, Λ: $\Lambda x = x\Lambda = x$

e) Ninguna cadena tiene inverso: dada $x \in E^*$ $(x \neq \Lambda)$, no existe ningún $y \in E^*$ tal que $xy = \Lambda$

Por consiguiente, $<E^*, >$, es decir, E^* con la operación de concatenación (que se representa, simplemente, poniendo una cadena a continuación de otra), es una estructura algebráica - de tipo monoide, a la que se suele llamar *monoide libre generado por E*.

El monoide es conmutativo sólo en el caso de que E - - conste de un solo elemento. Así, si E = {a}, entonces E^* = {Λ, a,aa, aaa, aaaa, ...}, y, por ejemplo, (aa) (aaa) = (aaa)(aa)= = aaaaa. Sin embargo, si E = {0,1}, entonces E^* = {Λ,0,1,00, - 01,10,11,000,001...}, y, por ejemplo, (10) (111) ≠ (111) (10).

En un autómata pueden considerarse el *monoide libre de entrada*, $<E^*, >$ y el *monoide libre de salida*, $<S^*, >$

También podemos olvidarnos de la cadena vacía, es de-- cir, en lugar de con E^*, trabajar con $E^+ = E^* - \{\Lambda\}$. En este - caso deberíamos hablar del *semigrupo libre generado por E*, ya que no existe elemento neutro. Sin embargo, lo más frecuente - es trabajar con E^*.

3. COMPORTAMIENTO DE UN AUTÓMATA

3.1. Otra definición de autómata

En el Capítulo 1 comenzamos hablando de los autómatas como dispositivos que producen cadenas de símbolos a la salida en respuesta a cadenas de símbolos presentadas a la entrada. Según esto, podríamos definir un autómata como una fun- - ción:

$$F^*: E^* \to S^* \qquad [3.1.1.]$$

que hace corresponder a cada cadena de entrada, $x \in E^*$, una cadena de salida, $F^*(x) = y \in S^*$.

Ahora bien, vamos a ver que el autómata queda perfectamente definido restringiendo el rango de la función de S^* a S, es decir, podemos definir un autómata como una función

$$F: E^* \to S \qquad [3.1.2.]$$

que hace corresponder a cada cadena de entrada, $x \in E^*$, el último símbolo obtenido como salida, $F(x) = s \in S$. En efecto, si $x = e_0\, e_1 \ldots e_{n-1}$, tendremos como símbolos de salida

$F(e_0)$ en el instante 1

$F(e_0 e_1)$ en el instante 2

$F(e_0 e_1 \ldots e_{n-1}) = F(x)$ en el instante n

Por consiguiente, la cadena de salida $F^*(x)$ se obtendrá concatenando todos estos símbolos:

$$F^*(x) = F^*(e_0 e_1 \ldots e_{n-1}) = F(e_0)\ F(e_0 e_1) \ldots F(e_0 e_1 \ldots e_{n-1}),$$

lo que nos demuestra que F^* queda determinada conociendo F.

Consideremos, por ejemplo, el autómata sumador binario serie estudiado en 1.4.2, y supongámoslo efectuando la suma 010110 + 011011 = 110001. En el instante t_0 recibirá por la entrada los bits de menor peso, es decir, $e_0 = 01$; en el instante t_1 tendremos $e_1 = 11$, etc. Así, la cadena de entrada será

$$x = 01.11.10.01.11.00$$

(Obsérvese que, en contra de lo que es habitual, utilizamos un punto para indicar la concatenación, a fin de evitar ambigüedades en este caso, ya que los símbolos de entrada están formados por dos símbolos de nuestro alfabeto ordinario. Obsérvese también que las cadenas se escriben de izquierda a derecha en el tiempo, con lo que resulta un orden de escritura inverso al habitual en aritmética).

En el instante t_0 tendremos como salida:

$$F(e_0) = F(01) = 1;$$

$$\text{en } t_1 : F(e_0e_1) = F(01.11) = 0;$$

$$\text{en } t_2 : F(e_0e_1e_2) = F(01.11.10) = 0;$$

$$\text{en } t_3 : F(e_0e_1e_2e_3) = F(01.11.10.01) = 0;$$

$$\text{en } t_4 : F(e_0e_1e_2e_3e_4) = F(01.11.10.01.11)=1;$$

$$\text{y en } t_5 : F(e_0e_1e_2e_3e_4e_5)=F(01.11.10.01.11.00)=1.$$

De modo que la cadena total de salida es:

$$F^*(x) = F(e_0)F(e_0e_1)\ldots F(e_0e_1\ldots e_6) = 100011,$$

que es el resultado de la suma escrito de izquierda a derecha según se van obteniendo los bits del resultado a partir del de menor peso.

La definición [3.1.2] considera al autómata exclusivamente desde el punto de vista de entrada-salida, es decir, como una "caja negra", a diferencia de la definición [1.1.1], - en la que se contempla lo que sucede en el interior de la "caja". Algunos autores, para resaltar la diferencia entre ambos, les dan distintos nombres, y así, por ejemplo, al autómata de-

finido por [3.1.2] le llaman "máquina", y al definido según - [1.1.1], "circuito", y este mismo convenio seguiremos nosotros en adelante cuando nos interese destacar que nos referimos a - una u otra definición.

En este punto, es natural que surjan unas preguntas inmediatamente: dada una máquina, ¿podemos encontrar su circuito?. Y, evidentemente, la inversa. Para responder a ellas se - hace precisa una consideración matemática previa sobre la definición [1.1.1]. En efecto, el dominio de las funciones f y g - es $E \times Q$, lo que quiere decir que estas funciones nos permiten obtener el estado siguiente y la salida conociendo el estado - actual y el *símbolo* de entrada. Para conocer la respuesta del circuito no a un símbolo, sino a una *cadena* de entrada es necesario ampliar el dominio a $E^* \times Q$

3.2. Ampliación del dominio de las funciones de un autómata

En el Apartado 1.3 vimos que el dominio de g podía ampliarse de $E \times Q$ a $\{E \cup \{\Lambda\}\} \times Q$ solamente si existe una función de salida $h: Q \to S$; en este caso decíamos que el autómata es - una máquina de Moore, y teníamos:

$$f(\Lambda,q) = q;\ g(\Lambda,q) = h\,[f(\Lambda,q)] = h(q)$$

Para extender ahora el dominio a $E^* \times Q$ basta con definir, para todo $x_1, x_2 \in E^*$,

$$f(x_1x_2,q) = f\,[x_2, f(x_1,q)]$$

$$g(x_1x_2,q) = g\,[x_2, f(x_1,q)] = h\,[f(x_1x_2),q]$$

3.3. El comportamiento de entrada-salida, o las máquinas definidas por un circuito.

Definición 3.3.1. Dado un autómata (circuito) A = <E, S,Q,f,g>, definimos el comportamiento de entrada-salida de A - inicializado en el estado q por la función

$$C_q : E^* \to S$$

que aplica a cada $x \in E^*$ un $s = g(x,q)$

Es decir, si en el instante t_0 el autómata se encuentra en el estado q e introducimos la cadena $x = e_1 e_2 \ldots e_n$, se obtendrán las salidas

$$C_q(e_1) \text{ en } t = t_0$$

$$C_q(e_1 e_2) \text{ en } t = t_0+1$$

$$C_q(e_1 e_2 \ldots e_n) = C_q(x) \text{ en } t = t_0+n-1$$

Vemos así que para todo autómata (circuito) pueden definirse, en principio, tantas funciones de la forma [3.1.2] - (es decir, tantas "máquinas") como estados tenga el autómata, aunque hay que advertir que algunas de estas funciones pueden ser idénticas entre sí (lo que correspondería a estados equivalentes, según la definición que daremos enseguida).

3.4. Equivalencia y accesibilidad

Damos a continuación una serie de definiciones cuyo - sentido se captará mejor con ayuda de ejemplos, que desarrollaremos en el Apartado 3.5.

Definición 3.4.1. Dados dos autómatas con los mismos alfabetos de entrada y salida, $A_1 = \langle E,S,Q_1,f_1,g_1\rangle$ y $A_2 = \langle E,S,Q_2,f_2,g_2\rangle$, $q_1 \in Q_1$ es equivalente a $q_2 \in Q_2$ si $C_{q_1} = C_{q_2}$.

La misma definición sirve para estados equivalentes dentro de un mismo autómata: basta considerar que $Q_1 = Q_2$, $f_1 = f_2$, $g_1 = g_2$.

Definición 3.4.2. Un autómata está en forma mínima (o es observable) si

$$(C_{q_1} = C_{q_2}) \rightarrow (q_1 = q_2)$$

Es decir, en un autómata en forma mínima no existen estados equivalentes.

Definición 3.4.3. Los autómatas $A_1 = \langle E,S,Q_1,f_1,g_1\rangle$ y $A_2 = \langle E,S,Q_2,f_2,g_2\rangle$ son equivalentes si

$$\{C_{q_1} \mid q_1 \in Q_1\} = \{C_{q_2} \mid q_2 \in Q_2\}$$

Definición 3.4.4. q_2 es accesible desde q_1 si existe $x \in E^*$ tal que $f(x,q_1) = q_2$.

Definición 3.4.5. El subautómata conectado de un autómata $A = \langle E,S,Q,f,g\rangle$ es $A^C = \langle E,S,Q^C,f^C,g^C\rangle$, con

$$Q^C = \{q_2 \in Q \mid \exists x \in E^*, \exists q_1 \in Q, f(x,q_1) = q_2\}$$

y f^C y g^C son las restricciones de f y g de $E^* \times Q$ a $E^* \times Q^C$.

Es decir, el subautómata conectado de un autómata está formado por todos los estados del autómata original que son accesibles desde algún otro estado.

Definición 3.4.6. Un autómata A es fuertemente conectado si $A = A^C$

3.5. Ejemplos

3.5.1. Autómata reconocedor de la cadena 010

Considérense los AF dados por los diagramas de Moore - de las figuras 2.12 y 2.13

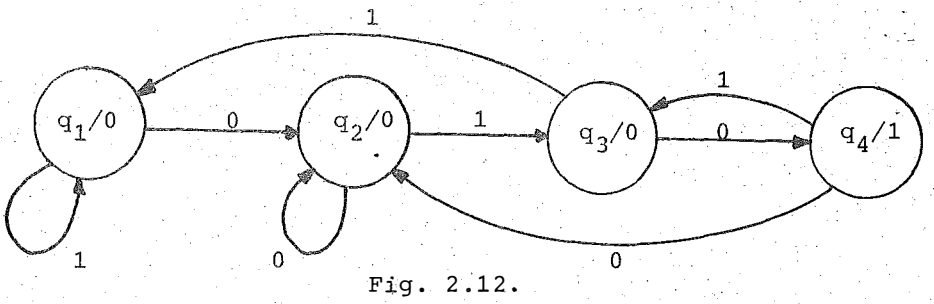


Fig. 2.12.

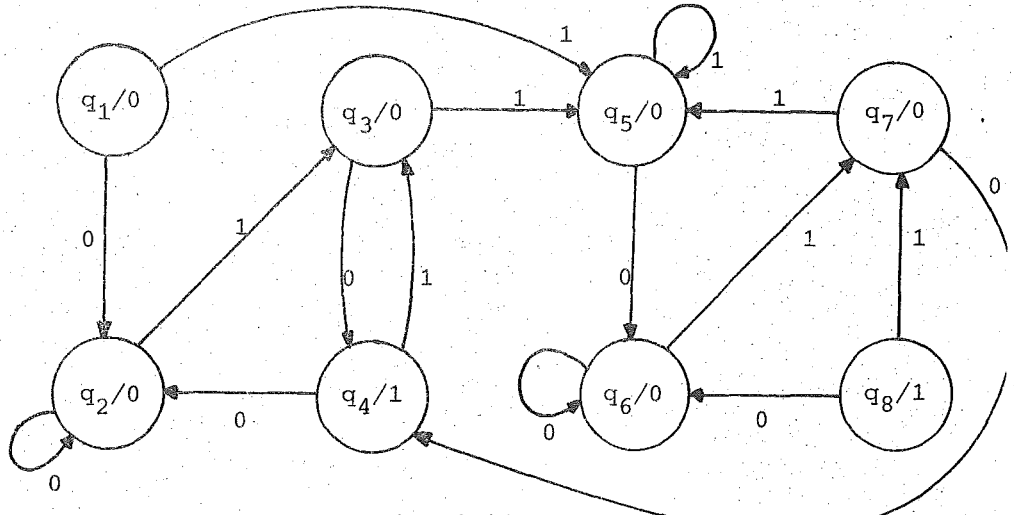


Fig. 2.13.

a) Calcular C_{q_1}, C_{q_2}, C_{q_3} y C_{q_4} (en la Fig. 2.12) para cadenas de entrada de longitud igual o inferior a 3.

b) Comprobar que el AF de la figura 2.12 está en forma mínima y es fuertemente conectado. ¿Ocurre lo mismo con el de la figura 2.13?.

c) Comprobar que ambos autómatas son equivalentes

Pasemos a resolver las cuestiones planteadas.

a) Basta con seguir, para cada estado inicial, las - - transiciones provocadas sucesivamente por cada símbolo de la - cadena (leída ésta de izquierda a derecha) y anotar la salida final. Resumimos los resultados en la tabla de la figura 2.14.

	Λ	0	1	00	01	10	11
C_{q_1}	0	0	0	0	0	0	0
C_{q_2}	0	0	0	0	0	1	0
C_{q_3}	0	1	0	0	0	0	0
C_{q_4}	1	0	0	0	0	1	0

	000	001	010	011	100	101	110	111
C_{q_1}	0	0	1	0	0	0	0	0
C_{q_2}	0	0	1	0	0	0	0	0
C_{q_3}	0	0	1	0	0	0	0	0
C_{q_4}	0	0	1	0	0	0	0	0

Fig. 2.14

b) Para comprobar que un AF está en forma mínima hay - que ver si el comportamiento es distinto para cada estado. Esto no es fácil con los conocimientos que tenemos hasta ahora, puesto que habría que ir ensayando con diferentes cadenas de - entrada hasta que se observe una diferencia de comportamiento entre dos estados para la misma cadena. Pero como E^* es infinito, si no encontramos tal diferencia después de un número finito de cadenas no podemos garantizar que el AF esté en forma reducida. (Empleando una terminología que se definirá con precisión en el Tema siguiente, no tenemos un algoritmo). Ahora bien, en el Apartado 5.3 demostraremos que basta con ensayar todas las cadenas x tales que lg (x) $\leq$ n-1 (n = número de estados); y si $C_{q_1}(x) \neq C_{q_2}(x)$ para esas cadenas podremos asegurar que $C_{q_1}(x) \neq C_{q_2}(x), \forall x \in E^*$. De este modo tendremos un algoritmo, ya que el número necesario de ensayos es finito.

En el autómata de la figura 2.12 se ve enseguida, con ayuda de la tabla de la figura 2.14, que

$$C_{q_1}(\Lambda) = 0,\ C_{q_2}(\Lambda) = 0,\ C_{q_3}(\Lambda) = 0,\ C_{q_4}(\Lambda) = 1,$$

por lo que q_4 no es equivalente a ninguno de los otros tres. Observando los comportamientos para entrada x = 0 deducimos - que q_3 tampoco es equivalente a ninguno, y, de igual modo, la no equivalencia de q_2 la deducimos de la entrada x = 10.

Por consiguiente, podemos asegurar que el AF de la figura 2.12 está en forma mínima. No ocurre lo mismo con el de - la figura 2.13. En efecto, si se van ensayando cadenas diferentes de entrada se irá viendo que $C_{q_1}(x) = C_{q_5}(x)$, $C_{q_2}(x) = C_{q_6}(x)$, $C_{q_3}(x) = C_{q_7}(x)$ y $C_{q_4}(x) = C_{q_8}(x)$. Se puede compro-

bar que esto ocurre para todas las cadenas de longitud igual o inferior a 7 (n = 8), por lo que q_1 es equivalente a q_5, q_2 a q_6 y q_4 a q_8, y, por consiguiente, el AF no está en forma mínima.

El carácter de fuertemente conectado se ve por simple inspección de los diagramas. En efecto, en la figura 2.12 vemos que cualquiera de los cuatro estados tiene por lo menos un arco que entra en él, es decir, todos los estados son accesibles desde algún otro, y, por tanto, el autómata es fuertemente conectado. Sin embargo, en la figura 2.13 podemos observar que ni q_1 ni q_8 son accesibles desde ningún otro estado, por lo que el autómata no es fuertemente conectado.

c) Ensayando todas las cadenas de longitud igual o inferior a 7 deducimos que q_1 y q_4 de la figura 2.13 son equivalentes a q_1 de la figura 2.12, q_2 y q_6 a q_2, q_3 y q_7 a q_3 y q_4 y q_8 a q_4. Por tanto, ambos AF son equivalentes.

Finalmente, justifiquemos el título puesto a este ejemplo, aunque los autómatas reconocedores serán objeto de estudio en el Capítulo 4. Puede comprobarse que siempre que en la cadena de entrada aparezca la sucesión 010 la salida del autómata es 1 al recibir el último símbolo de la sucesión (el segundo 0); en caso contrario la salida será 0, y no dará 1 hasta que en la cadena de entrada no vuelvan a aparecer seguidos un 0, un 1 y luego otro 0.

3.5.2. Ejemplo 2: Equivalencia máquina de Moore-Máquina de Mealy

En el Apartado 1.3 decíamos que para cualquier máquina de Mealy se puede obtener una máquina de Moore equivalente, y construíamos esta máquina escindiendo cada estado de la primi-

tiva en tantos estados como salidas pudieran asociársele. También decíamos que en lo sucesivo nos referiríamos siempre a máquinas de Moore, en las que podemos considerar el elemento neutro Λ en la entrada, y, por consiguiente, el monoide libre de entrada, E^*, mientras que en el caso de máquinas de Mealy tendríamos que trabajar con el semigrupo libre de entrada, E^+. Al definir la equivalencia entre autómatas (Def. 3.4.3) nos hemos basado en el comportamiento de entrada-salida (Def.3.3.1), que se ha definido no con E^+, sino con E^*, suponiendo implícitamente que los autómatas considerados son máquinas de Moore. Lo que pretendemos en este ejemplo es ver que la máquina de Moore construida a partir de una de Mealy como se indica en el Apartado 1.3 es efectivamente equivalente a ella, en el sentido de equivalencia de comportamiento. Para ello tenemos que ver que los conjuntos de los comportamientos de ambas máquinas son idénticos, con la salvedad de que no tiene sentido hablar de la entrada Λ; es decir, por esta sola vez, digamos que el comportamiento para un estado q es

$$C_q : E^+ \to S$$

Pues bien, sobre el ejemplo de las figuras 2.2 y 2.3, y si llamamos A a la primera y $\hat{A}$ a la segunda, podemos comprobar que

$$C_{q_1}(a) = \hat{C}_{q_1}(a) = 0;\ C_{q_1}(b) = \hat{C}_{q_1}(b) = 1;\ C_{q_1}(aa) =$$

$$= \hat{C}_{q_1}(aa) = 0;\ C_{q_1}(ab) = \hat{C}_{q_1}(ab) = 1;\ C_{q_1}(ba) = \hat{C}_{q_1}(ba) = 0;$$

etc.,

es decir,

$$C_{q_1} = \hat{C}_{q_1}$$

Análogamente se ve que $C_{q_2} = \hat{C}^0_{q_2} = \hat{C}^1_{q_2}$ y que $C_{q_3} = \hat{C}^0_{q_3} = \hat{C}^1_{q_3}$.

En general, lo que ocurre es que todos los estados q^s resultantes de una escisión de q tienen igual comportamiento - (salvo, naturalmente, para $x = \Lambda$), ya que se toma

$$\hat{g}(x,q^s) = g(x,q)$$

para todos los q^s

4. CAPACIDAD DE RESPUESTA DE UN AUTÓMATA FINITO

4.1. Introducción

Hemos visto que un circuito con n estados puede realizar hasta n máquinas diferentes. Cada una de estas máquinas - viene definida por el comportamiento de entrada-salida para - el correspondiente estado.

El número de cadenas diferentes en E^* es infinito. Pero como el autómata es finito es imposible que responda de - distinta manera a cada una de ellas. Es decir, debe ser posible particionar E^* en un número finito de subconjuntos tales que el autómata sea incapaz de distinguir dos cadenas pertenecientes al mismo subconjunto. Veremos que tales subconjuntos pueden considerarse como clases de equivalencia; natural-

mente, cuanto mayor sea el número de clases de equivalencia mayor será la capacidad del autómata para responder de distinta manera a cadenas diferentes.

Cuando decimos que el autómata "responde" a una cadena o "distingue" entre una u otra pensamos en el símbolo de salida asociado a cada cadena, de acuerdo con la definición [3.1.2]. Si consideramos exclusivamente máquinas de Moore, existirá una función de salida $h: Q \to S$. Vamos en este apartado a simplificar el análisis, suponiendo que h es biyectiva, es decir, que a cada estado podemos asignarle una salida diferente (*). En este caso, dos cadenas pertenecerán a la misma clase de equivalencia (serán indistinguibles para todas las máquinas definidas por el circuito) si, considerando un estado inicial cualquiera, el estado final es el mismo para ambas cadenas. Nos vemos así conducidos a estudiar, para cada cadena, funciones de la forma $Q \to Q$, y, si llamamos Q^Q al conjunto de todas estas funciones, el comportamiento global del autómata vendrá determinado por una función $K: E^* \to Q^Q$, que asigna a cada cadena de entrada una función $Q \to Q$. Si el número de estados diferentes es n, habrá n^n funciones $Q \to Q$, por lo que el número máximo de clases de equivalencia en E^* será n^n.

Vamos a formalizar estas ideas, y para ello comenzare

(*) Esta simplificación no resta generalidad al análisis. En efecto, lo que haremos será estudiar la "respuesta de estados" en el sentido de que para cada entrada nos fijaremos no en la salida, sino en las transiciones que provoca entre los estados. Si h no fuera biyectiva lo único que podría ocurrir es que dos cadenas diferentes en cuanto a su "respuesta de estados" fueran indistinguibles en cuanto a la salida, pero esto es fácil de analizar conociendo la función $h: Q \to S$.

mos por recordar algunos conceptos de álgebra.

4.2. Repaso de algunos conceptos de álgebra

4.2.1. Monoide de transformaciones de un conjunto

Una transformación t en un conjunto C se define como una función de C en sí mismo: $t: C \to C$.

Teorema 4.2.1.1. Sea C un conjunto cualquiera y sea $C^C = \{t: C \to C\}$ el conjunto de todas las transformaciones en C. Entonces $\langle C^C, \circ \rangle$, donde "$\circ$" representa la ley de composición de funciones t, es un monoide, llamado monoide de transformaciones de C.

La demostración es casi inmediata, teniendo en cuenta la evidencia de que la composición de funciones t es una operación cerrada y asociativa, es decir, si $t_a, t_b, t_c \in C^C$,

$$t_a \circ t_b \in C^C$$

y

$$t_a \circ (t_b \circ t_c((c) = (t_a \circ t_b) \circ t_c(c) = t_a(t_b(t_c(c))), \forall\, c \in C$$

Además, podemos definir un elemento neutro en C^C, $t_1: C \to C$, tal que $t_1 \circ t_i = t_i \circ t_1 = t_i$; este elemento neutro es $t_1(x) = x$. Por consiguiente, $\langle C^C, \circ \rangle$ cumple las condiciones para ser un monoide.

Ejemplo. Sea $C = \{0,1\}$. Veamos cuál es el monoide de transformaciones de C.

C^C tendrá cuatro elementos:

t_0: $t_0(0) = 0$, $t_0(1) = 0$

t_1: $t_1(0) = 0$, $t_1(1) = 1$

t_2: $t_2(0) = 1$, $t_2(1) = 0$

t_3: $t_3(0) = 1$, $t_3(1) = 1$

(Obsérvese que, acorde con la notación anterior, t_1 es el elemento neutro).

La composición de t_2 con t_3, por ejemplo, será:

$$t_2 \circ t_3(0) = t_2(t_3(0)) = t_2(1) = 0$$

$$t_2 \circ t_3(1) = t_2(t_3(1)) = t_2(1) = 0;$$

luego $t_2 \circ t_3 = t_0$

Análogamente pueden hallarse las demás composiciones; el resultado puede ponerse en forma tabular (tabla del monoide):

$\circ$	t_0	t_1	t_2	t_3
t_0	t_0	t_0	t_0	t_0
t_1	t_0	t_1	t_2	t_3
t_2	t_3	t_2	t_1	t_0
t_3	t_3	t_3	t_3	t_3

Esta tabla representa la operación $\circ$ en el conjunto C^C, y por tanto describe al monoide $\langle C^C, \circ \rangle$.

4.2.2. Homomorfismo entre monoides

Definición 4.2.2.1. Si $\langle S_1, * \rangle$ y $\langle S_2, . \rangle$ son dos semigrupos, una función $f: S_1 \to S_2$ decimos que es un homomorfismo entre ambos semigrupos si preserva la ley de composición interna, es decir, si

$$f(a*b) = f(a).f(b), \forall a,b \in S_1,$$

o, expresado gráficamente, si el diagrama

$$\begin{array}{ccc} S_1 \times S_1 & \xrightarrow{*} & S_1 \\ {\scriptstyle f \times f}\downarrow & & \downarrow{\scriptstyle f} \\ S_2 \times S_2 & \xrightarrow{\cdot} & S_2 \end{array}$$

es conmutativo

Definición 4.2.2.2. Un isomorfismo entre semigrupos es un homomorfismo en el que la función f es biyectiva

Definición 4.2.2.3. Si S_1 es un monoide con elemento neutro e_1, S_2 es un monoide con elemento neutro e_2, f es un homomorfismo (isomorfismo)entre S_1 y S_2, y se cumple que

$$f(e_1) = e_2,$$

entonces f es un homomorfismo (isomorfismo) monoide

Ejemplos

1) Si P = {1,2,3,...} y N = {0,1,2,3,....}, entonces - <P,.>, <N,+>y <N,.> son monoides, mientras que <P,+> es un semigrupo, pero no un monoide. (¿Por qué?). La multiplicación - por un número natural a en <N,+>, es decir, <N,+>→ <N,+>:n→a.n es un homomorfismo, ya que $a.(n_1+n_2) = a.n_1 + a.n_2$. Sin embargo, en general, no es un homomorfismo para <N, .>. (¿En qué casos particulares lo es?).

2) Llamemos R al conjunto de los números reales. <R,+> y <R, .> son monoides (¿Por qué?). La función

$$< R, + > \to < R, . >: r \to a^r \qquad (a, r \in R)$$

es un homomorfismo monoide, ya que

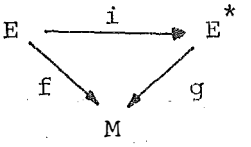
$$a^{r_1 + r_2} = a^{r_1} \cdot a^{r_2} \quad \text{y} \quad 0^r = 1$$

(¿Es isomorfismo?).

4.2.3. Monoide libre y homomorfismo

Hemos definido en el Apartado 2.5 el monoide libre generado por un alfabeto E, $<E^*, >$

Teorema 4.2.3.1. Sea i: $E \to E^*$ la función que aplica todo elemento de E en la correspondiente cadena de longitud - unidad, es decir, i(a) = a, $\forall$ a $\in$ E, y sea f cualquier función de E en el conjunto de cualquier monoide <M, *>. Entonces, - existe un único homomorfismo monoide g: $<E^*, > \to <M, *>$ tal que g ∘ i = f, es decir, tal que el diagrama



es conmutativo.

Demostración.

Para cadenas de longitud 1 podemos definir g(a) = f(a) para que se satisfaga f(a) = g∘i(a) = g(i(a)) = g(a). Si x es una cadena de longitud l ≥ 2 podemos descomponerla en x = y a_n, donde lg(y) = l-1 y lg(a_n) = 1, y tendremos:

$$g(x) = g(y\ a_n) = g(y) * g(a_n)$$

para que m sea un homomorfismo. Como lg(a_n) = 1,

$$g(x) = g(y) * f(a_n)$$

Del mismo modo, podemos descomponer y en z.a_{n-1}, y así sucesivamente, con lo que, por inducción sobre la longitud de la cadena, si x = $a_1a_2.....a_n$ podemos determinar g(x) así:

$$g(x) = f(a_1) * f(a_2).....*f(a_n)$$

Finalmente, si e es el elemento neutro de <M, *>, haremos g(Λ) = e para obtener un homomorfismo monoide.

*Este teorema nos permite extender el dominio de cualquier función E → M de alfabeto E en el conjunto de un monoide <M, *> a un homomorfismo monoide <E^*, > → <M, *>, y nos será de*

utilidad más adelante.

4.2.4. Relaciones de congruencia y monoide cociente.

Definición 4.2.4.1. Una relación binaria R en un conjunto C es un subconjunto de C x C. Si c_1 y c_2 son elementos de C, decimos que c_1 y c_2 están relacionados por R si $(c_1,c_2) \in R$, y generalmente se escribe: $c_1 \; R \; c_2$

Definición 4.2.4.2. R es una relación de equivalencia en un conjunto C si y sólo si R es una relación binaria que tiene las propiedades:

- reflexiva : $c \; R \; c, \; \forall c \in C$
- Simétrica : $(c_1 \; R \; c_2) \rightarrow (c_2 \; R \; c_1), \forall c_1,c_2 \in C$
- Transitiva: $(c_1 R c_2) \wedge (c_2 R c_3) \rightarrow (c_1 R c_3), \forall c_1,c_2,c_3 \in C$

Definición 4.2.4.3. Dados un conjunto C y una relación de equivalencia R, la clase de equivalencia de $c \in C$ módulo R es $[c]_R = \{x | x \; R \; c\}$. (En adelante, siempre que no se preste a ambiguedad, suprimimos el subíndice R). El conjunto cociente C/R es el conjunto de las clases de equivalencia de C módulo R. El número de elementos de C/R es el índice de la equivalencia.

Así, una relación de equivalencia origina una división de C en subconjuntos disjuntos, es decir, una partición.

Definición 4.2.4.4. Dado un monoide <M, *> y una relación de equivalencia, R, en M, R es una relación de congruencia en <M, *> si aRb implica que (a*c) R (b*c) y (c*a) R (c*b) para todo $c \in M$.

Teorema 4.2.4.5. Si R es una relación de congruen--

cia en el monoide $< M, * >$, el conjunto cociente $M/R = \{[a] | a \in M\}$ con la operación "." definida por

$$[a] . [b] = [a*b]$$

es un monoide

Demostración

En primer lugar hay que demostrar que la operación "." está bien definida sobre las clases de equivalencia, es decir, que el resultado es independiente de que se tome un miembro u otro de la clase. Para ello, sean $a_1 \in [a]$, $a_2 \in [a]$, $b_1 \in [b]$, $b_2 \in [b]$. Tenemos pues que $a_1 R a_2$ y $b_1 R b_2$, y, como R es una relación de congruencia, podemos escribir:

$$(a_1*b_1) \; R \; (a_1*b_2) \text{ y } (a_1*b_2) \; R \; (a_2*b_2),$$

y, por la propiedad transitiva de R,

$$(a_1*b_1) \; R \; (a_2*b_2), \text{ es decir, } [a_1*b_1] = [a_2*b_2]$$

lo que indica que $[a] . [b]$ está bien definida.

Para ver que $<M/R,.>$ es un monoide, basta con comprobar dos hechos:

1) "." es asociativa. En efecto, como "*" es asociativa (pues $<M, *>$ es un monoide),

$$[a] . \{[b] . [c]\} = [a] . [b*c] = [a*(b*c)] = [(a*b)*c] =$$

$$= \{[a] . [b]\} . [c]$$

2) Existe un elemento neutro. En efecto, si el elemento neutro en $<M, * >$ es e, tendremos que

$$[a] \cdot [e] = [a*e] = [a]$$

y

$$[e] \cdot [a] = [e*a] = [a] .$$

por lo que $[e]$ es el elemento neutro para $<M/R, .>$

Definición 4.2.4.6. El monoide $<M/R,.>$ se llama monoide cociente de M por R.

4.3. Comportamiento de entrada-estados

Sea un AF

$$A = <E,S,Q,f,g>$$

Sabemos que dados un símbolo de entrada y un estado podemos obtener el estado siguiente mediante la función

$$f: E \times Q \to Q$$

Dado solamente un símbolo de entrada, $e \in E$, podemos definir una función que aplique a cada estado el siguiente bajo esa entrada determinada, es decir, una función de Q en Q, $k(e):Q \to Q$. Existirá entonces una función

$$k: E \to Q^Q,$$

siendo Q^Q el conjunto de las funciones de Q en Q, que sabemos por el Teorema 4.2.1.1 que, con la composición de funciones, es un monoide. La función k se determina a partir de f del siguiente modo: para cada $e \in E$, y cada $q \in Q$, $[k(e)](q)=f(e,q)$

Ahora bien, por el Teorema 4.2.3.1, la función k puede extenderse a un homomorfismo entre monoides:

$$K: \langle E^*, \rangle \to \langle Q^Q, \circ \rangle,$$

con $K(e_1e_2.....e_n) = k(e_1) \circ k(e_2) \circ ..... \circ k(e_n)$. Llamaremos a K comportamiento de entrada-estados del autómata.

Ejemplo. Consideremos el autómata detector de paridad descrito por el diagrama de la figura 2.6. La función $k:\{0,1\} \to Q^Q$ se obtiene fácilmente del diagrama y se puede resumir en forma de tabla:

		Estado siguiente	
		k(0)	k(1)
Estado inicial	q_1	q_1	q_2
	q_2	q_2	q_1

Calculemos K para algunas cadenas:

$K(0) = k(0)$; $K(1) = k(1)$;

$K(00) = k(0) \circ k(0) = k(0)$

$K(01) = k(0) \circ k(1) = k(1)$

$K(10) = k(1) \circ k(0) = k(1)$

$K(11) = k(1) \circ k(1) = k(0)$

etc.

En general,

$K(x) = k(0)$ si se tiene un número par de unos

$K(x) = k(1)$ si se tiene un número impar de unos

Si la cadena es vacía, $K(\Lambda)$ será la función identidad en Q, es decir, la función que aplica q_1 en q_1 y q_2 en q_2, que coincide con $k(0)$.

4.4. Relación equirrespuesta y monoide de un autómata

Sea un autómata A con comportamiento de entrada-estados K.

Definición 4.4.1. La relación equirrespuesta de A, $\cong$ es una relación binaria en E^* tal que

$$x \cong y \text{ si y sólo si } K(x) = K(y), \ \forall x, y \in E^*$$

$\cong$ es una relación de equivalencia, ya que se cumple:

$$K(x) = K(x), \ \forall x \in E^*$$

$$[K(x) = K(y)] \rightarrow [K(y) = K(x)], \forall x, y \in E^*$$

$$[K(x) = K(y)] \wedge [K(y) = K(z)] \rightarrow [K(x) = K(z)] \ \forall x,y,z \in E^*$$

Además es una relación de congruencia en el monoide $\langle E^*, \ \rangle$. En efecto, si $x \cong y$, $K(x) = K(y)$, y $K(xz) = K(x) \circ K(z) =$

$K(y) \circ K(z) = K(yz)$. Análogamente, $K(zx) = K(zy)$. Por tanto, según el Teorema 4.2.4.5, $\langle E^*/\cong, \ \rangle$, es decir, el conjunto cociente $E^*/\cong$ con la operación de concatenación, es un monoide.

Definición 4.4.2. Dado un autómata con un comportamiento de entrada-estados K que origina una relación equirespuesta $\cong$ en E^*, el monoide cociente $\langle E^*/\cong, \ \rangle$ se llama monoide del autómata.

Si el número de estados es n, el monoide del autómata tendrá como máximo n^n elementos. En efecto, el número de transformaciones en el conjunto Q, es decir, de funciones diferentes $Q \to Q$ es n^n. El homomorfismo K aplica entonces un conjunto infinito, E^*, en un conjunto, Q^Q, que tiene n^n elementos. Si esta aplicación es suprayectiva la relación equirrespuesta tendrá índice n^n, es decir, inducirá n^n clases de equivalencia en E^* (si la aplicación no fuera suprayectiva, es decir, si existieran una o más funciones $Q \to Q$ a las que no correspondiese ningún elemento de E^*, el número de clases de equivalencia sería menor). Por consiguiente, el número máximo de elementos del conjunto cociente $E^*/\cong$ es n^n.

El monoide de un autómata refleja la capacidad de éste para responder de distinto modo a las cadenas de entrada. En efecto, en E^* hay infinitas cadenas, mientras que en $E^*/\cong$ hay como máximo n^n elementos, que son las clases de congruencia de $\cong$. Si dos cadenas diferentes, x e y, están en la misma clase, es decir, $x \cong y$, entonces $K(x) = K(y)$, es decir, el homomorfismo K las aplica sobre el mismo elemento de Q^Q, o lo que es lo mismo, ambas producen la misma transformación $Q \to Q$, y el autómata será incapaz de distinguir una de la otra.

4.5. Ejemplos

4.5.1. Detector de paridad

Ya hemos estudiado el comportamiento de entrada-estados del detector de paridad, llegando a la conclusión de que sólo hay dos clases de equivalencia en E^*; en efecto, veíamos que $K(x) = k(0) = k(\Lambda)$, si x tiene un número par de unos (o ninguno) $K(x) = k(1)$, si x tiene un número impar de unos. Luego toda cadena x es equivalente a "Λ" o a "1". Llamamos $[\Lambda]$ y $[1]$ a estas dos clases de equivalencia, que serán los elemen-

tos del monoide del autómata. Al tener un número finito de elementos, podemos describirlo mediante una tabla que indique el elemento resultante de la concatenación de otros dos:

	[Λ]	[1]
[Λ]	[Λ]	[1]
[1]	[1]	[Λ]

4.5.2. Reconocedor de la cadena 010

El autómata, representado en la figura 2.12, tiene 4 estados, por lo que el número total posible de transformaciones en Q, es decir, el número máximo de elementos del monoide del autómata es $4^4 = 256$. Veamos si se tienen todos ellos. Para ello, calculemos K(x) para cadenas de longitud creciente - desde x = Λ. Resumimos los resultados en la tabla de la figura 2.15. El procedimiento para construir una tabla de este tipo es el siguiente:

Tenemos tantas filas como estados tiene la máquina. - Para cada cadena x ponemos en una columna los estados resul-- tantes de la función K(x): Q → Q. Comenzamos por Λ, para là - que la función es la unidad (es decir, q_1 en q_1, q_2 en q_2, - etc.). Para las cadenas de longitud 1 (en este caso 0 y 1) miramos en el diagrama de Moore; en este ejemplo, cuando se - - aplica x = 0 la imagen de q_1 es q_2, la de q_2 es q_2, etc. Te-- niendo ya K(x) para cadenas de longitud 1 no hace falta consultar más el diagrama; por ejemplo, K(01) = K(0) ∘ K(1), y así, con 0, q_1 se aplica en q_2, y luego con 1 q_2 se aplica en q_3, por lo que la imagen de q_1 será q_3. Para cadenas de longitud 3 nos basamos en los resultados anteriores; por ejemplo, - K(010) = K(01) ∘ K(0) = K(0) ∘ K(10) (las dos formas de hacerlo son válidas). La tabla se va así rellenando por columnas -

estado final

estado inicial	x	Λ	0	1	00	01	10	11	000	001	010
	q_1	q_1	q_2	q_1	q_2	q_3	q_2	q_1	q_2	q_3	q_4
	q_2	q_2	q_2	q_3	q_2	q_3	q_4	q_1	q_2	q_3	q_4
	q_3	q_3	q_4	q_1	q_2	q_3	q_2	q_1	q_2	q_3	q_4
	q_4	q_4	q_2	q_3	q_2	q_3	q_4	q_1	q_2	q_3	q_4

x	011	100	101	110	111	0100	0101
q_1	q_1	q_2	q_3	q_2	q_1	q_2	q_3
q_2	q_1	q_2	q_3	q_2	q_1	q_2	q_3
q_3	q_1	q_2	q_3	q_2	q_1	q_2	q_3
q_4	q_1	q_2	q_3	q_2	q_1	q_2	q_3

Fig. 2.15

aumentando la longitud de las cadenas por postconcatenación de los símbolos del alfabeto a las cadenas ya tratadas, hasta que encontramos cadenas x tales que $K(x) = K(y)$, donde $lg(y) \leq lg(x)$; entonces podemos asegurar que $x \cong y$, y no es preciso que sigamos aumentando x, ya que, si la concatenamos con un símbolo cualquiera, e, tendremos $K(xe) = K(ye)$, es decir, $xe \cong ye$, con $lg(ye) \leq lg(x)$, por lo que la clase de ye o bien ya ha aparecido, (si $lg(y) < lg(x)$) o bien va a aparecer (si $lg(y) = lg(x)$).

Concretemos con nuestro ejemplo. Siete de las ocho cadenas de longitud 3 tienen el mismo comportamiento que cadenas

de longitud 2: K(000) = K(00); K(001) = K(01), etc., por lo - que [000] = [00]; [001] = [01], etc. La única cadena cuyo comportamiento difiere de los anteriores es 010. Por tanto, estudiamos 0100 y 0101, y vemos que K(0100) = K(00) y K(0101) = = K(001). Nos quedan así solamente 8 clases de equivalencia: [Λ], [0], [1], [00], [01], [10], [11] y [010]. El monoide del autómata tendrá pues 8 elementos, no 256:K dista mucho de ser suprayectiva. La tabla del monoide se obtiene sin dificultad de la figura 2.15, y es la representada en la figura 2.16; por ejemplo, para saber el resultado de [11] [10], con ayuda de la figura 2.15 vemos que K(1110) = K(11) ∘ K(10) = K(00), por lo que [11] [10]= [00]

	[Λ]	[0]	[1]	[00]	[01]	[10]	[11]	[010]
[Λ]	[Λ]	[0]	[1]	[00]	[01]	[10]	[11]	[010]
[0]	[0]	[00]	[01]	[00]	[01]	[010]	[11]	[010]
[1]	[1]	[10]	[11]	[00]	[01]	[00]	[11]	[010]
[00]	[00]	[00]	[01]	[00]	[01]	[10]	[11]	[010]
[01]	[01]	[010]	[11]	[00]	[01]	[00]	[11]	[010]
[10]	[10]	[00]	[01]	[00]	[01]	[010]	[11]	[010]
[11]	[11]	[00]	[11]	[00]	[01]	[00]	[11]	[010]
[010]	[010]	[00]	[01]	[00]	[01]	[010]	[11]	[010]

Fig. 2.16

Con la tabla del monoide se puede hallar rápidamente el comportamiento para cualquier cadena. Así, por ejemplo, si queremos saber $C_q(010000110)$, tendremos:

$$[010000110] = [010]\ [00]\ [01]\ [10] = [00]\ [00] = [00],$$

y como [00] transforma cualquier estado inicial en q_2 (Fig.2.15), y q_2 da salida 0, tendremos que $C_{q_i}(0100000110) = 0$, $i = 1,2,3,4$

4.5.3. Contador en código binario natural módulo 10

Un contador es un autómata que proporciona en cada momento información sobre el número de entradas de un determinado tipo recibidas hasta ese momento. Supondremos que $E = \{0,1\}$, y que el contador debe dar como salida el código en BCD natural correspondiente al número total de 1 recibidos. Si el autómata es finito sólo podrá contar hasta un determinado número y a partir de él volver al principio; un contador módulo p contará de 0 a p-1.

Consideremos p = 10, es decir, el contador contará de 0 a 9 y al recibir el décimo 1 volverá a contar desde 0. Cada vez que recibe un 1 deberá cambiar de estado (y, naturalmente, de salida). Así es fácil ver que el diagrama de Moore es el de la figura 2.17, en donde se supone que el estado inicial es q_0, y como símbolos de salida se han tomado $s_0 \ldots\ldots s_9$ para abreviar la notación (en realidad serían 0000, 0001....1001).

Como hay 10 estados, el monoide del autómata podría tener hasta 10^{10} elementos. Ahora bien, si construimos la tabla de K(x) (Fig. 2.18) vemos que sólo hay 10 clases de equivalencia.

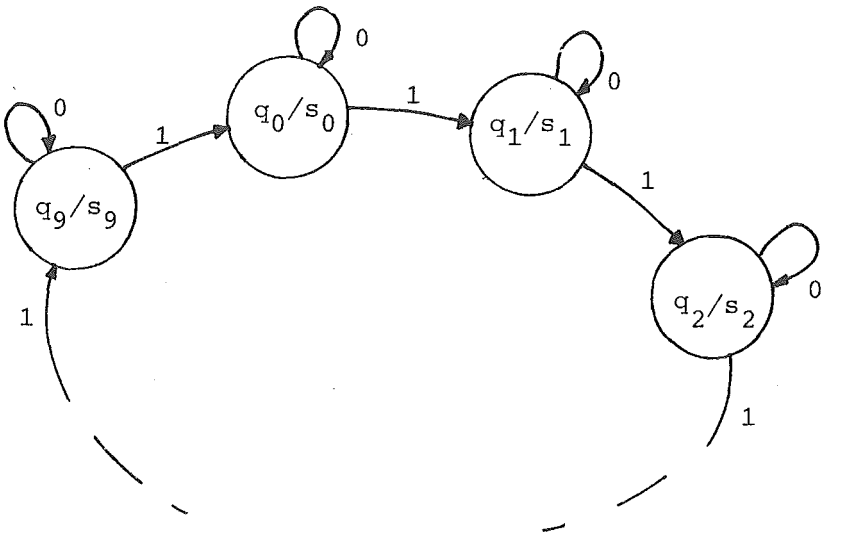


Fig. 2.17.

	Λ	0	1	1^2.....1^9	1^{10}
q_0	q_0	q_0	q_1	q_2 q_9	q_0
q_1	q_1	q_1	q_2	q_3 q_0	q_1
.	.	.	.	. .	.
:	:	:	:	: :	:
:	:	:	:	: :	:
q_8	q_8	q_8	q_9	q_0 q_7	q_8
q_9	q_9	q_9	q_0	q_1 q_8	q_9

Fig. 2.18.

En efecto, vemos que $0 \cong \Lambda$, por lo que sólo hay que considerar cadenas compuestas por 1. Calculamos $K(1^2)$, $K(1^3)$, etc. (el exponente indica concatenación: $1^3 = 111$, etc.), y vemos que $1^{10} \cong \Lambda$. Por consiguiente, las clases de equivalencia son:

$$[\Lambda],\ [1],\ [1^2],\ [1^3],\ \ldots.,\ [1^9]$$

Dejamos como ejercicio al lector la obtención de la tabla del monoide del autómata. Con ella podrá fácilmente comprobar que cualquier cadena que contenga n_0 "ceros" y n_1 "unos" - en cualquier orden pertenece a la clase de equivalencia $[1^m]$, donde $m = n_1 \pmod{10}$.

4.6. Relación entre el comportamiento de entrada-salida y el comportamiento de entrada-estados.

El comportamiento de entrada-salida lo definíamos para un estado inicial q:

$$C_q: E^* \to S$$

Así, para cada entrada, x, $C_q(x)$ es la función que aplica x - en g(x,q):

$$C_q(x): x \to g(x,q)$$

El comportamiento de entrada-estados, sin embargo, se define para todo el conjunto de estados:

$$K: E^* \to Q^Q$$

De modo que, para cada entrada, x, K(x) aplicará x en una función de transformación entre estados:

$$K(x): x \to 1:Q \to Q,$$

donde la función l está determinada por la función f restringida a la entrada x:

$$f: E \times Q \to Q,$$

$$f(x): Q \to Q$$

El sentido físico es el siguiente: C_q *determina, dado un estado inicial, el símbolo final de salida para cada cadena de entrada; K determina, para cada cadena de entrada, la correspondiente transformación entre estados*, es decir, nos dice que de q_1 se pasará a q_i, de q_2 a q_j, etc. Por tanto, K nos proporciona más información acerca del comportamiento interno del autómata. Por otra parte, la relación de equivalencia en E^* inducida por K y el monoide del autómata resultante nos permiten deducir fácilmente C_q para cualquier cadena de entrada. Así, en el ejemplo del reconocedor de la cadena 010 veíamos que $[010000100] = [00]$, y de aquí que $C_{q_i} = 0$; para llegar a la misma conclusión con el procedimiento que seguíamos en el Apartado 3.5 hubiera sido preciso reconocer sobre el diagrama (Fig. 2.12) todas las transiciones producidas por la cadena.

Por otra parte, podría pensarse en estudiar el comportamiento de entrada-salida para todo el conjunto de estados. Para ello podemos definir un comportamiento global de entrada salida, $K_S(x)$ como una aplicación de E^* en el conjunto de funciones l: $Q_0 \to S_T$ donde Q_0 son los estados iniciales y S_T las salidas finales. K_S no será ahora un homomorfismo entre monoides, como lo era K ya que el conjunto de funciones l no es un monoide; por tanto, K_S no nos permite definir un monoide de la máquina, y por ello hemos utilizado K en el análisis anterior. En cualquier caso, K_S también nos define una relación de equivalencia en E^*, que podría llamarse "relación equisalida", $\cong_S$, tal que $x \cong_S y$ si y sólo si $K_S(x) = K_S(y)$. $\cong_S$ parti-

ciona E^* en clases de equivalencia. Ahora bien, suponiendo máquinas de Moore, tenemos una función $h: Q_T \to S_T$ que nos aplica el estado final en la salida final correspondiente a ese estado, y por tanto, si llamamos $[x]_i$ a la clase de equivalencia de $E^*/\cong$ que conduce a una determinada transformación $k_i: Q_0 \to Q_T$ y $[y]_j$ a la clase de equivalencia de $E^*/\cong_S$ que corresponde a una determinada $l_i: Q_0 \to S_T$, tendremos:

$$[y]_j = \bigcup [x]_i \text{ para todo } i \text{ tal que } k_i \circ h = l_j$$

Esto demuestra que las clases de equivalencia de $\cong_S$ están perfectamente determinadas por $\cong$ y h, y justifica lo que decíamos en nota a pie de página en el Apartado 4.1: que al considerar sólo la respuesta de los estados el estudio no pierde generalidad.

5. MINIMIZACIÓN DE UN AUTÓMATA FINITO

5.1. Planteamiento del problema

En el ejemplo 3.5.1 veíamos dos AF equivalentes, uno de los cuales estaba en forma mínima. Si estos AF describen un sistema secuencial que debe realizarse físicamente escogeremos el que está en forma mínima, ya que el coste de la realización, como se verá en el Capítulo 3, crece con el número de estados. Es, pues, importante poder saber si un AF está en forma mínima, y, si no lo está, hallar un AF en forma mínima equivalente a él. Comenzaremos por ver que éste existe siempre; a continuación veremos que para detectar equivalencia entre estados no es preciso realizar infinitos ensayos con cadenas de entrada, y, finalmente, veremos un algoritmo para mini

mizar un AF, es decir, para hallar otro AF en forma mínima -- equivalente.

Ante todo, debemos señalar que la equivalencia entre estados de un AF, definida por

$$q_1 \equiv q_2 \text{ si y sólo si } C_{q_1}(x) = C_{q_2}(x), \forall x \in E^*$$

es una relación de equivalencia en el conjunto Q, pues, como es inmediato comprobar, es reflexiva, simétrica y transitiva. Por consiguiente, puede definirse un conjunto cociente, $Q/\equiv$, cuyos elementos serán las clases de equivalencia: $[q]$ será la clase que contiene a q.

5.2. Autómata en forma mínima de un autómata dado

Vamos a demostrar que, dado $A = \langle E,S,Q,f,h\rangle$, el AF definido por $A_M = \langle E,S,Q_M,f_M,h_M\rangle$, donde

$$Q_M = Q/\equiv$$

$$f_M(x,[q]) = [f(x,q)]$$

$$h_M([q]) = h(q),$$

es equivalente a A y está en forma mínima.

En primer lugar, habrá que demostrar que f_M y h_M están bien definidas, es decir, que son independientes del elemento q que se tome dentro de la clase $[q]$, o, lo que es lo mismo, que si $q_1 \equiv q_2$, entonces $[f(x,q_1)] = [f(x,q_2)] \forall x$, y

$h(q_1) = h(q_2)$. En efecto, por la definición de la equivalencia, si $q_1 \equiv q_2$, $g(x,q_1) = g(x,q_2)$ $\forall\, x \in E^*$. Si tomamos $x = \Lambda$ resulta $h(q_1) = h(q_2)$. Y si tomamos $x = x_1x_2$,

$$g(x_1x_2,q_1) = g(x_2,f(x_1,q_1))$$

$$g(x_1x_2,q_2) = g(x_2,f(x_1,q_2)),$$

por lo que $f(x_1,q_1) \equiv f(x_1,q_2)$ $\forall\, x_1$, es decir, $[f(x,q_1)] = [f(x,q_2)]$ $\forall\, x \in E^*$.

Por otra parte, es evidente que A y A_M son equivalentes, puesto que todo estado q_i de A será equivalente al estado $[q_i]$ de A_M.

Finalmente, A_M está en forma mínima, ya que si los estados $[q_1]$ y $[q_2]$ de A_M fueran equivalentes, q_1 y q_2 de A deberían ser también equivalentes, por lo que $[q_1] = [q_2]$.

5.3. Comprobación de la equivalencia entre estados de un autómata.

5.3.1. Teorema de las particiones sucesivas

Teorema 5.3.1.1. Si $P_0,P_1,P_2,\ldots$ es una secuencia infinita de particiones en un conjunto finito Q tal que, para todo k, se cumple:

a) P_{k+1} es un refinamiento de P_k, es decir, todo bloque de P_{k+1}, B^i_{k+1}, está contenido en un bloque, B^j_k, de P_k: $B^i_{k+1} \subset B^j_k$

b) $(P_{k+1} = P_k) \rightarrow (P_{k+2} = P_{k+1})$,

Entonces existe un número entero $k_0 < \text{card}\ (Q)$ tal - que $P_k = P_{k_0}$, $\forall\, k \geq k_0$

Para demostrarlo, supongamos que card $(P_0) = 0$, card $(P_1) = 1$, card $(P_2) = 2$.... Pero como, para todo k, card $(P_k) \leq$ card $(Q) = n$ (finito), deberá existir un entero $k_0 < n$ tal que card $(P_{k_0+1}) =$ card (P_{k_0}), es decir, $P_{k_0+1} = P_{k_0}$, y, por la condición b), $P_k = P_{k_0}$, $\forall\, k \geq k_0$.

5.3.2. Equivalencia de orden k entre estados

Definición 5.3.2.1. Dos estados de un AF son equivalentes de orden k si y sólo si conducen a la misma salida para cadenas de entrada de longitud igual o inferior a k:

$$(q_1 \equiv_k q_2) \leftrightarrow ((\lg(x) \leq k) \rightarrow C_{q_1}(x) = C_{q_2}(x))$$

Obsérvese que para comprobar la equivalencia de orden k ya no hay que hacer un número infinito de comprobaciones o "experimentos".

Teorema 5.3.2.2. Dado un AF y $q_1, q_2 \in Q$, existe un $k_0 < \text{card}\ (Q)$ tal que q_1 y q_2 son equivalentes $(q_1 \equiv q_2)$ si y sólo si q_1 y q_2 son equivalentes de orden k_0 $(q_1 \equiv_{k_0} q_2)$

Para demostrar este teorema nos apoyaremos en el - - 5.3.1.1. Veamos pues que las particiones inducidas en Q por - las sucesivas equivalencias de orden 0,1,...k, k+1... cumplen las condiciones a) y b) de aquel teorema.

a) Si $q_1 \equiv_{k+1} q_2$ entonces $C_{q_1}(x) = C_{q_2}(x)$ $\forall\, x \leq k+1$, y, evidentemente, $\forall\, x \leq k$, por lo que $q_1 \equiv_k q_2$. Por tanto, todo bloque de la partición P_{k+1} inducida por $\equiv_{k+1}$ está contenido en un bloque de P_k, es decir, P_{k+1} es un refinamiento de P_k.

b) Supongamos que $P_{k+1} = P_k$, es decir, $(q_1 \equiv_k q_2) \rightarrow (q_1 \equiv_{k+1} q_2)$. Si q_1 y q_2 son equivalentes de orden k+1, los estados $f(e,q_1)$ y $f(e,q_2)$ serán, sea cual sea e, equivalentes de orden k, pero como $P_{k+1} = P_k$ también serán equivalentes de orden k+1, y por ello q_1 y q_2 son equivalentes de orden k+2, de donde $P_{k+2} = P_{k+1}$. (El razonamiento es una sucesión de condicionales. Definiendo, para abreviar la notación, las variables proposicionales

a: $P_{k+1} = P_k$

b: $P_{k+2} = P_{k+1}$

c: $q_1 \equiv_k q_2$

c': $q_1 \equiv_{k+1} q_2$

c": $q_1 \equiv_{k+2} q_2$

d: $f(e,q_1) \equiv_k f(e,q_2)$ $\forall\, e \in E$

d': $f(e,q_1) \equiv_{k+1} f(e,q_2)$ $\forall\, e \in E$

podemos expresarlo formalmente así:

$$[(a \rightarrow (c \rightarrow c')) \wedge (c' \rightarrow d) \wedge (a \rightarrow (d \rightarrow d') \wedge (c' \rightarrow c")) \rightarrow$$

$$\rightarrow ((c' \rightarrow c") \rightarrow b)] \rightarrow (a \rightarrow b),$$

que puede comprobarse que es una tautología).

En definitiva, se cumplen las condiciones a) y b) del Teorema 5.3.1.1, y, por tanto, existe $k_0 < \text{card}(Q) = n$ tal que $P_k = P_{k_0}$, $k \geq k_0$, con lo que $(q_1 \equiv_{k_0} q_2) \rightarrow (q_1 \equiv q_2)$

La conclusión importante es que, sin necesidad de conocer k_0, como $k_0 < n$ (número de estados del AF), para comprobar si dos estados son equivalentes bastará con comprobar si son equivalentes de orden n-1

5.4. Algoritmo para la minimización de un autómata finito.

El algoritmo para hallar el AF en forma mínima de un AF dado se basa en los resultados anteriores:

1. $k = 0$. Formar P_0, poniendo en el mismo bloque los estados que tengan asociada la misma salida ($q_i \equiv_0 q_j$ si y sólo si $h(q_i) = h(q_j)$).

2. Formar P_{k+1}, teniendo en cuenta que dos estados estarán en el mismo bloque de P_{k+1} si y sólo si para cada entrada los estados siguientes están en el mismo bloque de P_k ($q_i \equiv_{k+1} q_j$ si y sólo si $f(e,q_i) \equiv_k f(e,q_j)$, $\forall e \in E \cup \{\Lambda\}$).

3. Si $P_{k+1} \neq P_k$, incrementar k en una unidad y volver al paso 2.

4. $k_0 = k$. Proceso terminado. El AF en forma mínima tiene como estados $Q_M = Q/\equiv_{k_0}$.

5.5. Ejemplos

5.5.1. Ejemplo 1

Consideremos el autómata reconocedor de 010 descrito en 3.5.1, y partamos del diagrama no mínimo (Fig. 2.13). A efectos de aplicación del algoritmo es más cómodo representar el AF por la tabla de transiciones:

	0	1
$q_1/0$	q_2	q_5
$q_2/0$	q_2	q_3
$q_3/0$	q_4	q_5
$q_4/1$	q_2	q_3
$q_5/0$	q_6	q_5
$q_6/0$	q_6	q_7
$q_7/0$	q_4	q_5
$q_8/1$	q_6	q_7

1. Observando las salidas asociadas a cada estado podemos poner:

$$P_0 = \{q_1, q_2, q_3, q_5, q_6, q_7\}, \{q_4, q_8\}$$

2. Con entrada 0, de q_1, q_2, q_5 y q_6 se pasa a estados del primer bloque de P_0, y con 1 también. De q_3 y q_7, con 0 se pasa a estados del segundo bloque de

P_0, y con 1 a otros del primer bloque. Finalmente, de q_4 y q_8, con 0 se pasa al primer bloque de P_0 y con 1 también. Luego

$$P_1 = \{q_1,q_2,q_5,q_6\},\ \{q_3,q_7\},\ \{q_4,q_8\}$$

2'. En lugar de con palabras, pongamos simbólicamente los resultados:

$f(0,q_1) = q_2$
$f(0,q_2) = q_2$
$f(0,q_5) = q_6$
$f(0,q_6) = q_6$ } (1^{er}.bloque)

$f(1,q_1) = q_5$ (1^{er}.bloque)
$f(1,q_2) = q_3$ (2° bloque)
$f(1,q_5) = q_6$ (1^{er} bloque)
$f(1,q_6) = q_7$ (2° bloque)

$f(0,q_3) = q_4$
$f(0,q_7) = q_4$ } (3^{er}.bloque)

$f(1,q_3) = q_5$
$f(1,q_7) = q_5$ } (1^{er} bloque)

$f(0,q_4) = q_2$
$f(0,q_8) = q_6$ } (1^{er}.bloque)

$f(1,q_4) = q_3$
$f(1,q_8) = q_7$ } (2° bloque)

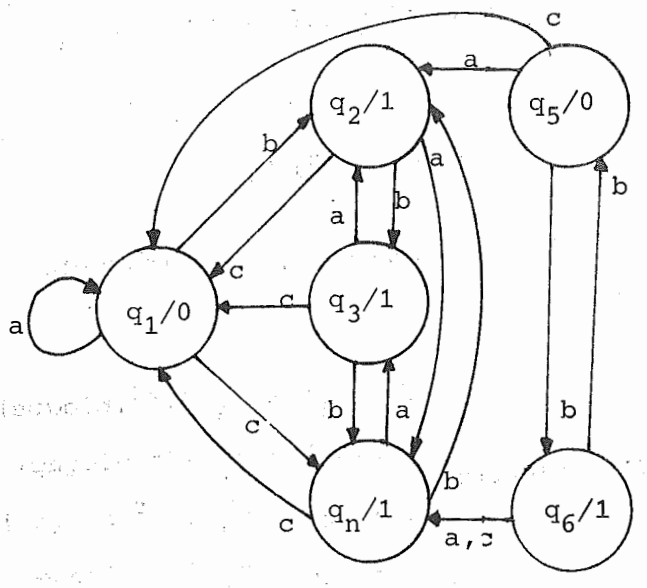
Luego

$$P_2 = \{q_1,q_5\}\ ,\ \{q_2,q_6\},\ \{q_3,q_7\},\ \{q_4,q_8\}$$

2". Si repetimos la operación, viendo a qué bloques de P_2 se pasa con cada entrada veremos que $P_3 = P_2$. - Luego $k_0 = 2$, y el AF minimizado tiene como esta-- dos los bloques definidos en P_2; dando los nuevos nombres $\{q_1,q_5\} = q_1$,etc., se obtiene el diagrama de la figura 2.12.

5.5.2. Ejemplo 2.

Sea el AF dado por el diagrama y la tabla de la figura 2.19.



	a	b	c
$q_1/0$	q_1	q_2	q_4
$q_2/1$	q_4	q_3	q_1
$q_3/1$	q_2	q_4	q_1
$q_4/1$	q_3	q_2	q_1
$q_5/0$	q_2	q_6	q_1
$q_6/1$	q_4	q_5	q_4

Fig. 2.19.

Siguiendo el algoritmo, encontramos sucesivamente:

$$P_0 = \{q_1, q_5\}, \{q_2, q_3, q_4, q_6\}$$

$$P_1 = \{q_1\}, \{q_5\}, \{q_2, q_3, q_4\}, \{q_6\}$$

$$P_2 = P_1$$

Llamando $q_1 = \{q_1\}$; $q_2 = \{q_2, q_3, q_4\}$; $q_3 = \{q_5\}$, y $q_4 = \{q_6\}$ tenemos el diagrama de la figura 2.20, que representa el AF en forma mínima equivalente al dado

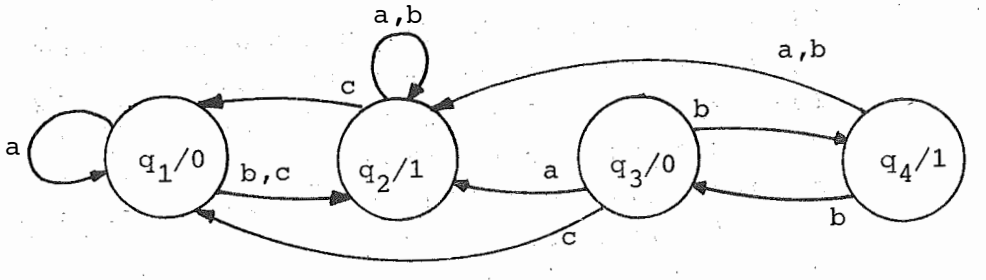


Fig. 2.20

6. RESUMEN

Hemos expuesto la teoría básica de los autómatas finitos procurando buscar la mayor generalidad, generalidad que se conserva aunque el estudio se restrinja a máquinas de Moore.

El comportamiento de entrada-salida es un concepto importante en relación con los lenguajes, que utilizaremos en el Capítulo sobre autómatas reconocedores. Otros conceptos importantes son los de equivalencia entre estados y entre autómatas, pero el interés de éstos radica más bien en la realización tecnológica.

El estudio del comportamiento global del autómata, es decir, inicializado en cualquier estado, nos ha llevado a establecer el concepto de monoide del autómata, estructura algebráica que refleja su capacidad para responder de distinto modo a cadenas diferentes de entrada.

Finalmente, hemos visto cómo se puede comprobar la equivalencia entre estados con un número finito de experimentos, lo que nos ha permitido establecer un algoritmo para obtener el AF en forma mínima equivalente a un AF dado.

7. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

El primer estudio riguroso sobre autómatas fue el publicado por MOORE (1.956). Con anterioridad, debido al desarrollo de los primeros ordenadores, se habían estudiado diversos métodos para la síntesis de circuitos secuenciales (HUFFMAN, 1.954; MEALY, 1.955). A finales de los años 50 se comenzó a ver la utilidad de los autómatas en relación con los len

guajes, y la mayor parte de los trabajos sobre teoría de autó matas finitos se realizó durante los años 60. Por esta razón, las referencias más importantes que pueden darse sobre este - campo son libros publicados entre 1.965 y 1.970. El de HARRI- SON (1.965) cubre tanto la parte combinacional como los circui tos secuenciales, con un tratamiento muy riguroso y fácil de - seguir, aunque se limita a estudiar autómatas reconocedores. - Otra obra recomendable es la de BOOTH (1.967), que, además de autómatas finitos, trata también las máquinas de Turing, len- guajes artificiales y autómatas estocásticos. Muchas de las - definiciones y terminología de este Capítulo las hemos tomado de ARBIB (1.969),que cubre muy ampliamente los diversos estu- dios desarrollados hasta aquella fecha. La parte que trata del monoide del autómata está basada en el Capítulo sobre monoides de GILBERT (1.976), libro muy interesante sobre aplicaciones - del álgebra moderna.

Para no alargar excesivamente el Tema hemos presenta- do el algoritmo de minimización reducido al caso de máquinas de Moore (la máquina de Mealy mínima tendrá, normalmente, me- nos estados). El caso general viene expuesto en cualquiera de los libros citados en éste o el siguiente Capítulo.

8. EJERCICIOS.

1. Obtener la tabla de transiciones y el diagrama de Moore de la máquina de Moore equivalente a la de Mealy descrita por la siguiente tabla:

q \ e	e_1	e_2
q_1	q_3/s_1	q_2/s_2
q_2	q_4/s_2	q_3/s_2
q_3	q_4/s_1	q_2/s_2
q_4	q_2/s_2	q_4/s_1

2. Definir las máquinas de Mealy y de Moore de un restador binario.

3. (SCALA y MINGUET, 1.974).

Una señal semafórica se acciona manualmente. Tiene tres estados caracterizados por los colores verde, amarillo y rojo (V,A,R). Como entrada tiene dos botones: uno de cierre (c) y otro de apertura (a) del semáforo.

El funcionamiento es de la siguiente manera: al pulsar una vez el botón de cierre el semáforo pasa de verde a amarillo; si se pulsa por segunda vez, pasa de amarillo a rojo; si en este estado se insiste en pulsar el botón de cierre, el semáforo permanece rojo. El botón de apertura, en todo caso, pone el semáforo verde.

Dibujar el diagrama de Moore llamando Q = {V,A,R} y E = {a,c}.

4. (SCALA y MINGUET, 1.974).

Un autómata finito admite como entradas E = {a,b,c}, que se pueden imaginar alimentadas por 3 teclas. Se encuentra en un estado inicial q_0. Como salida tiene dos luces: ver-

de (v) y roja (r). Mientras las letras se alimentan en orden alfabético, pudiéndose repetir una de ellas tantas veces como se quiera, el autómata deberá emitir un destello verde. Cuando se altere el orden alfabético, el autómata - deberá emitir un destello rojo y volver al estado inicial. La serie de letras que se alimenta puede empezar por cualquiera de ellas. Establecer los estados del autómata, el - diagrama de Moore y la tabla de transiciones.

5. Obtener los monoides del sumador y del restador binario y compararlos.

6. Un autómata retardador es aquel en que

$$s(t) = e(t - n)$$

Suponiendo E = S = {0,1} y n = 2, obtener el diagrama de Moore y el monoide del autómata.

7. Obtener el monoide del autómata cuya tabla de transición es:

e / q/s	a	b	c
$q_1/1$	q_2	q_2	q_3
$q_2/0$	q_1	q_2	q_3
$q_3/1$	q_3	q_3	q_2

8. (GILBERT, 1.976). En primavera, un brote de planta requiere unas condiciones adecuadas para su desarrollo. En una - determinada especie, el brote necesita que se presente un día de lluvia seguido de dos días calurosos sin ser interrumpido por ningún día frío o de helada. Además, si hay -

un día de helada después de que se haya desarrollado el brote, éste muere. Dibujar el diagrama de Moore de este proceso.

Puede utilizarse E = {L,C,F,H}, donde L quiere decir - "día lluvioso", C, "caluroso", etc., y S = {T,B,M}, donde - T = "latente", B = "brote", M = "muerto". ¿Cuál es el número de elementos en el monoide de este autómata?.

9. (GILBERT, 1.976). Un perro puede estar tranquilo, irritado, asustado, o irritado y asustado, en cuyo caso muerde. Si le damos un hueso se queda tranquilo. Si le quitamos uno de - sus huesos se pone irritado, y, si ya estaba asustado, nos muerde. Si le amenazamos se asusta y, si ya estaba irritado nos muerde. Obtener el diagrama de Moore y el monoide del - perro.

10. Deducir un diagrama de Moore para un autómata reconocedor de la cadena "321", suponiendo que E = {1,2,3}. Obtener el monoide del autómata. Comprobar si está en forma mínima.

11. Considérese un autómata finito definido por

$$E = \{a,b\};\quad S = \{0,1\};\quad Q = \{q_1,q_2,q_3,q_4\}$$

y la tabla de transiciones:

e / q	a	b
q_1	$q_2/1$	$q_1/0$
q_2	$q_3/0$	$q_4/0$
q_3	$q_3/0$	$q_1/1$
q_4	$q_3/0$	$q_1/1$

1°. Este autómata, ¿es una máquina de Moore o de Mealy?. ¿Por qué?.

2°. Si es una máquina de Melay, dibujar el diagrama de Moore - de la máquina de Moore equivalente.

3°. Minimizar el autómata resultante de la pregunta anterior.

4°. Si se ha seguido el algoritmo de minimización expuesto aquí se habrá obtenido la máquina de Moore en forma mínima. ¿Existe una máquina de Mealy equivalente que tenga menos estados?. Si es así, dibujar su diagrama de Moore.

CAPITULO 3.

CIRCUITOS SECUENCIALES

1. LA REALIZACIÓN DE AUTÓMATAS FINITOS

Los AF que hemos estudiado en el Capítulo anterior no son, en definitiva, más que modelos matemáticos para sistemas digitales con memoria. Ya mencionábamos en el Capítulo 1 el doble interés de su estudio: teórico (herramienta matemática para formalizar el estudio de los lenguajes) y práctico (realización de sistemas digitales que cumplen funciones especificadas).

La realización de un sistema digital puede enfocarse de dos maneras:

a) Puede construirse físicamente el sistema, partiendo de elementos sencillos.

b) Puede utilizarse un ordenador, que es un sistema digital de uso general, y programarse para que efectúe las funciones que se desean.

Por ejemplo, si el autómata que se quiere realizar es un ordenador, se seguirá, normalmente, el enfoque a). (También puede seguirse el enfoque b), y en ese caso se trataría de simulación o emulación en otro ordenador). Si el autómata es muy complicado (como es un ordenador), se descompondrá generalmente en subautómatas; uno de ellos podría ser el sumador binario serie, ya estudiado a nivel teórico, del que veremos su realización en el Apartado 6.1. (Conviene señalar que

los ordenadores suelen utilizar otro tipo de sumador, el paralelo, que exige unos circuitos más complicados, pero que es mucho más rápido).

Otro ejemplo puede ser el reconocimiento de cadenas. Hemos visto un AF que reconoce la cadena "010". Podemos pensar en un AF que reconozca varias cadenas, dando una salida diferente para cada una. Así, si E = {A,B,....Z} y S = {00000, 00001,..., 11111}, podría dar s = 00000 para x = PAR, s=00001 para x = CMP, s = 00010 para x = PAC, y así, sucesivamente, todos los códigos binarios de operación en EIT-2 para los correspondientes códigos en ENSAM. Este AF sería parte de un AF más completo, llamado "ensamblador de ENSAM", que traduciría a lenguaje de máquina los programas escritos en ENSAM. Pues bien, los autómatas ensambladores, caso particular de procesadores de lenguajes, no se construyen físicamente, sino mediante un programa de ordenador.

En este Capítulo nos vamos a dedicar especialmente a las técnicas para realizar autómatas finitos físicamente. La realización del AF, salvo raras excepciones, es con tecnología electrónica o electromecánica, y el sistema digital con memoria resultante se llama circuito secuencial.

2. ELEMENTOS DE UN CIRCUITO SECUENCIAL

2.1. Tipos de elementos

La realización de las funciones definidas por un AF implica dos tipos de elementos:

a) Elementos combinacionales, que realizan las funciones lógicas en las que no interviene el tiempo. Por

ejemplo, la función h: Q → S debe dar la salida -- asociada a cada estado, sin intervención del tiempo, por lo que es una función lógica puramente combinacional, representable por una tabla de verdad y realizable con los métodos estudiados en el Tema "Lógica".

b) Elementos con memoria, para realizar las funciones en las que interviene el tiempo. Así, f: E x Q → Q debe dar, para unos valores dados de E y Q en el instante t el valor resultante de Q en t + 1 (recordemos que este "1" se refiere a una escala de tiempos arbitraria), es decir, debe calcular el valor resultante de Q y memorizarlo para darlo en t + 1.

Pasemos a estudiar el funcionamiento de algunos de estos elementos.

2.2. Elementos combinacionales

Son las puertas lógicas NOT, AND, OR, NAND, NOR, etc., ya estudiadas, cuyo funcionamiento debe ser bien conocido. Realizadas generalmente mediante tecnología electrónica, para aplicaciones especiales aún se utilizan otras tecnologías: electromecánica, hidráulica, neumática, etc.

2.3. Elementos con memoria

2.3.1. Líneas de retardo

El elemento con memoria más sencillo (desde el punto

de vista conceptual) es aquel en que se tiene E=Q=S = {0,1} y la salida en cualquier instante es igual al estado en ese instante e igual a la entrada retardada un intervalo de tiempo θ:

$$s(t + \theta) = q(t + \theta) = e(t),$$

con e,q,s ∈ {0,1}. Este sencillo modo de funcionamiento se puede ilustrar gráficamente con un cronograma como el de la figura 3.1.

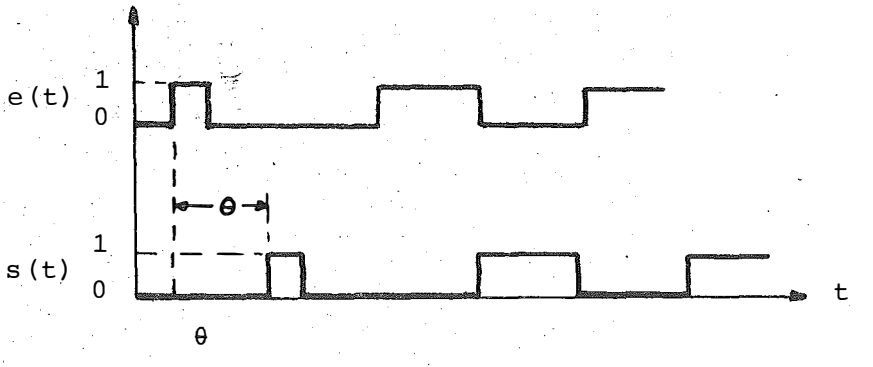


Fig. 3.1.

Estos elementos (que, de una manera general, se llaman líneas de retardo) pueden realizarse mediante líneas de retardo acústicas, líneas de transmisión, redes de condensadores e inductancias, puertas lógicas conectadas en serie, etc., y también con biestables, como veremos enseguida.

2.3.2. Biestables.

Los elementos de memoria más utilizados en los circuitos secuenciales son los biestables electrónicos. Un biestable es un circuito con dos estados estables que son también su salida y que, simbólicamente, se denominan "0" y "1" (en la realidad, cada uno de ellos corresponderá a un determinado

nivel de tensión). "Estables" quiere decir que el elemento sólo cambia de estado bajo la acción de las entradas. Hay muy diversos tipos de biestables, que difieren entre sí por los posibles símbolos de entrada y las funciones de transición. Todos ellos tienen dos líneas de salida: en una de ellas, denominada habitualmente "Q", se tiene el nivel de tensión correspondiente en cada momento al estado, 0 ó 1; en la otra, $\bar{Q}$, se tiene su complemento. Pasemos a ver el funcionamiento de algunos biestables.

Biestable tipo SR ("set-reset"). En este biestable $E = \{00,01,10\}$. Como las señales en los circuitos secuenciales son binarias, tendrá dos líneas de entrada, S y R (Fig. 3.2 a). Su funcionamiento puede expresarse formalmente, mediante la función de transición:

$$Q(t+1) = S(t) + \bar{R}(t).Q(t) \qquad (R.S = 0),$$

(aquí la unidad de tiempo es el intervalo que tarda el biestable en cambiar de estado o "tiempo de basculamiento"), o gráficamente, mediante el diagrama de Moore (Fig. 3.2 b), o mediante un cronograma (Fig. 3.2 c). Expresado con palabras, el funcionamiento es:

- $S = 0$, $R = 0$ hace que el biestable no cambie de estado.
- $S = 1$, $R = 0$ pone a 1 ("set") el biestable.
- $S = 0$, $R = 1$ lo pone a 0 o repone ("reset")
- $S = 1$, $R = 1$ es una entrada no permitida

Es fácil diseñar, con los métodos conocidos del Tema "Lógica", un circuito lógico que realice al biestable RS: basta minimizar la función de transición, función de S,R y Q y realimentar Q(t+1) a la entrada. Un posible circuito es el de la -

figura 3.2 d.

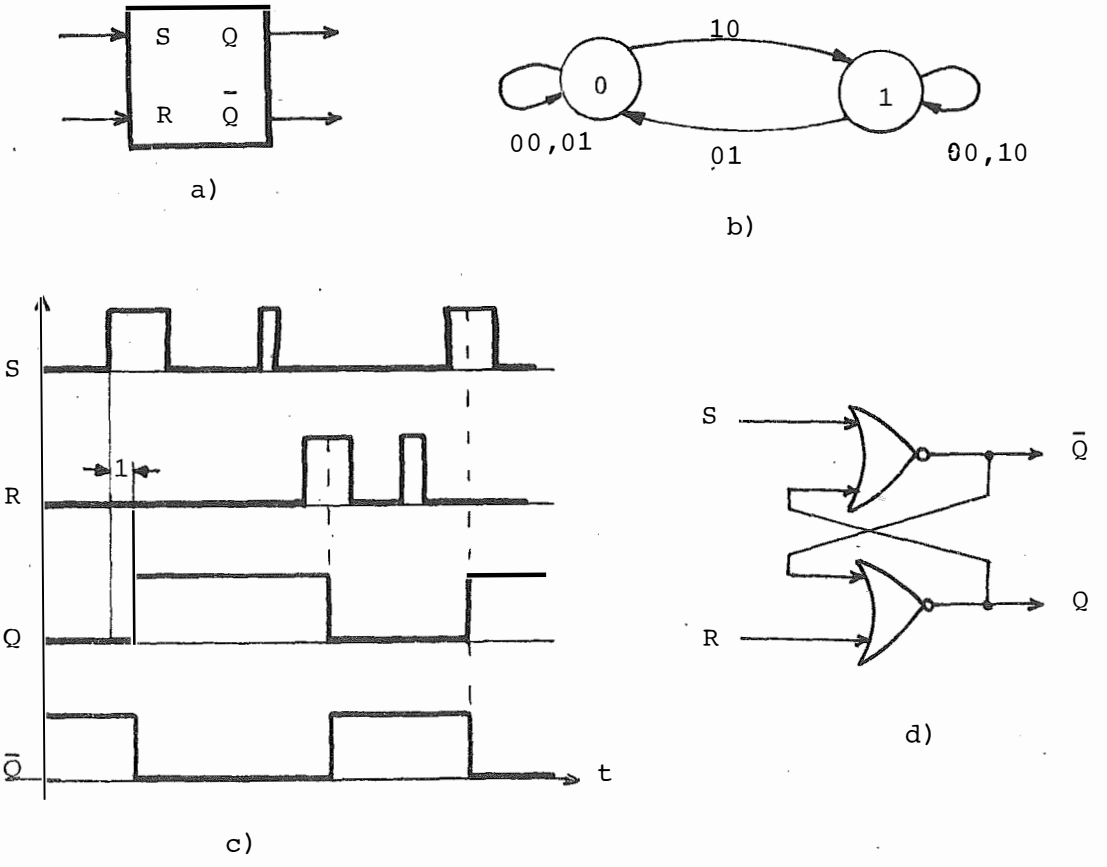
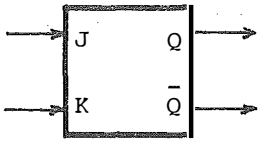


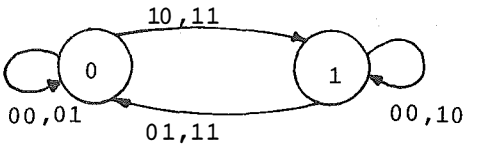
Fig. 3.2.

Biestable tipo JK. Es como el SR, sólo que aquí la entrada J = 1, K = 1 está permitida, y su efecto consiste en cambiar el estado que tuviera anteriormente el biestable (Fig. 3.3; en el cronograma, para no complicar demasiado su interpretación, se ha despreciado el tiempo de basculamiento).

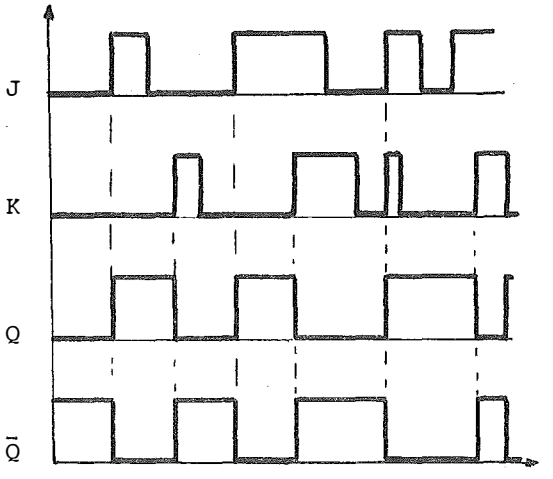
Hay que advertir que este biestable no se utiliza nunca con la realización de la Fig. 3.3.d, sino sincronizado, como se verá más adelante. La razón es que si se mantienen las entradas J = K = 1 el circuito se convierte en un oscilador con una frecuencia que depende del tiempo de basculamiento del biestable.



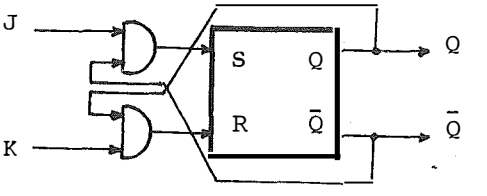
a)



b)



c)



d)

Fig. 3.3.

Biestable tipo D. También tiene dos líneas de entrada, ya que E = {00,01,10,11}. Estas líneas se llaman D ("data") y C ("clock") (Fig. 3.4 a). Su funcionamiento puede representarse formalmente así:

$$Q(t+1) = D(t).C(t) + Q(t).\bar{C}(t).$$

que corresponde al diagrama de la figura 3.4 b. Ahora bien, aquí hay una diferencia muy importante en cuanto al significado de los símbolos "0" y "1" para D y para C. Para D, "0" significa, por ejemplo, una tensión de 0 v., y "1", 5 v.; sin embargo, para C "1" significa que hay una variación de tensión de 0 v. a 5 v., y "0" que no hay tal variación. En otras palabras, el valor de C es cero salvo en el instante en que su -

tensión cambia de nivel bajo a nivel alto. Véase la figura - 3.4 c para ilustrar este significado de "0" y "1".

Un cronograma que ilustra el funcionamiento del biestable tipo D es el de la figura 3.4 d, donde hemos también - despreciado el tiempo de basculamiento. Obsérvese que si los cambios en D van retardados ligeramente un intervalo constante $\varepsilon \ll \theta$ con relación a los cambios de 0 a 1 de C, lo que hace - el biestable es exactamente retardar la entrada D en θ segundos, siendo θ el período del reloj (C).

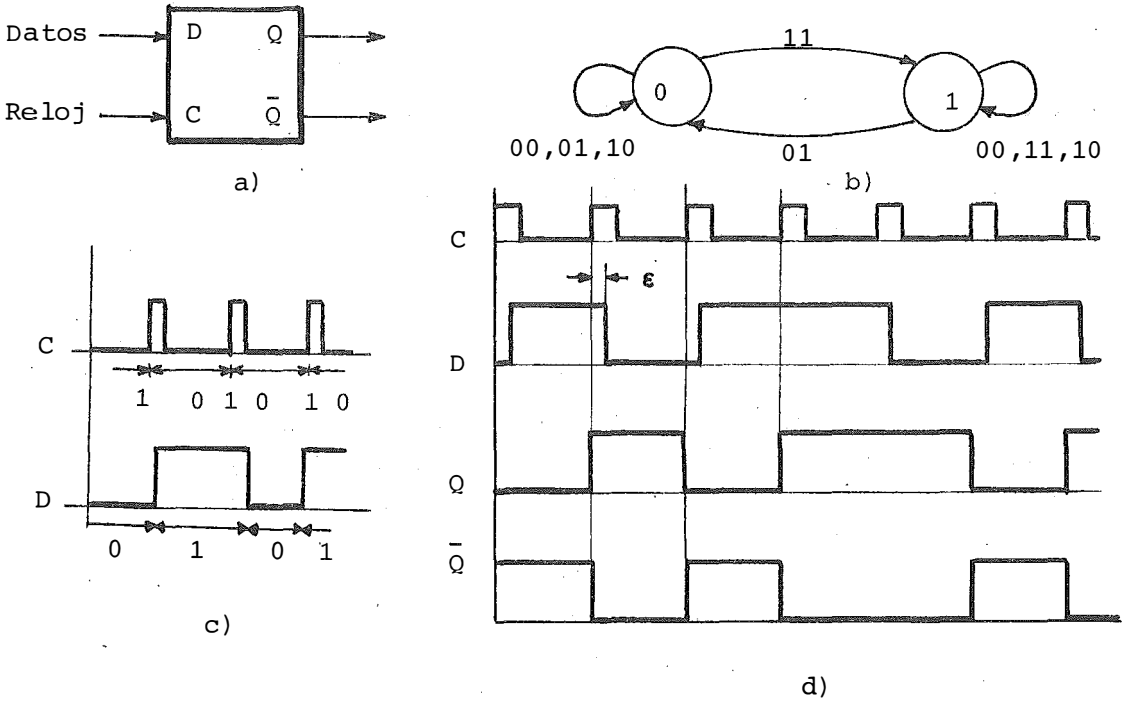


Fig. 3.4.

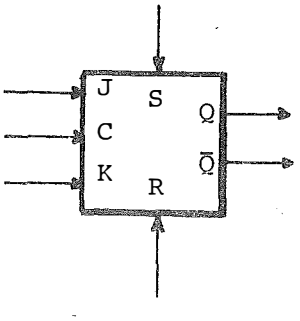
Este tipo de biestable, que sólo cambia de estado - - cuando cambia el nivel de la señal de reloj se llama sincronizado por flancos ("edge triggered"). El que hemos visto está sincronizado por el flanco de subida; también lo hay sincronizado por el flanco de bajada (es decir, cambia de estado cuan

do C pasa de 1 a 0). No entramos ya en su circuitería lógica, que el lector interesado puede consultar en la bibliografía, y que, en cualquier caso, el alumno estudiará en la asignatura - de Electrónica Digital.

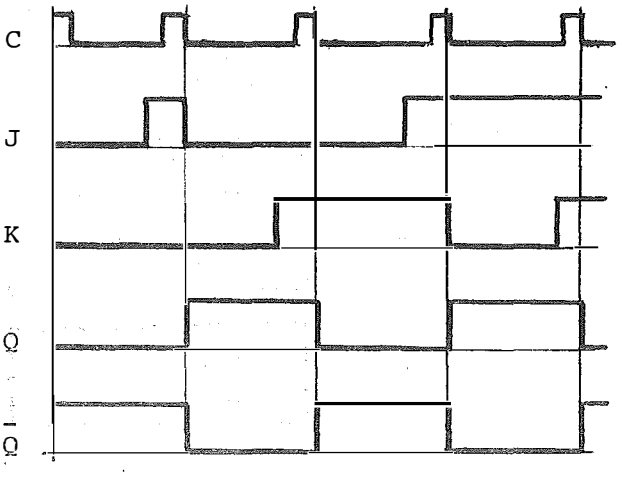
Biestable tipo JK sincronizado. Tiene 5 líneas de entrada : S,R,J,K y C(Fig. 3.5 a). S y R se utilizan para poner lo a 1 ó a 0 de una manera asíncrona, es decir, independientemente de C, mientras que J y K son la puesta a 1 ó a 0 síncronas. La ecuación que describe su comportamiento es

$$Q(t+1) = S + \bar{R}\,\bar{K}\,Q + \bar{R}\,\bar{C}\,Q + \bar{R}\,J\,C\,\bar{Q}$$

Aquí el valor lógico "1" para C significa una doble - transición nivel bajo-nivel alto-nivel bajo. Esto es debido a la constitución interna, en la que no entramos, que incluye - dos biestables en una configuración llamada "maestro-esclavo". Por ello, las transiciones de la señal de salida, Q, se dan - en el flanco de bajada de C. El cronograma de la figura 3.5 b ilustra el funcionamiento síncrono (S y R actúan como en un - S R normal, y, en caso de conflicto, predominan sobre las JK)



a)



b)

Fig. 3.5.

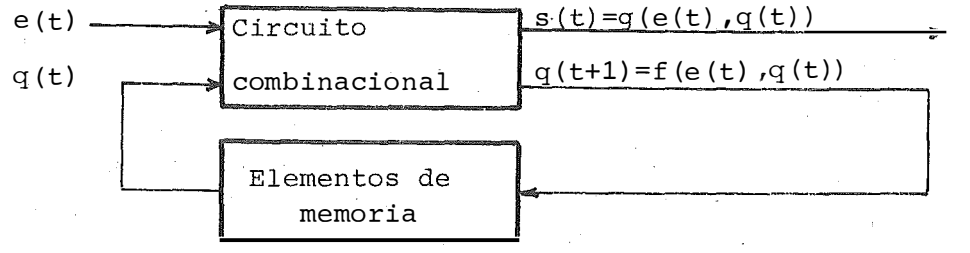
3. MODELOS BÁSICOS DE CIRCUITOS SECUENCIALES.

Recordemos que las funciones que definen un AF son

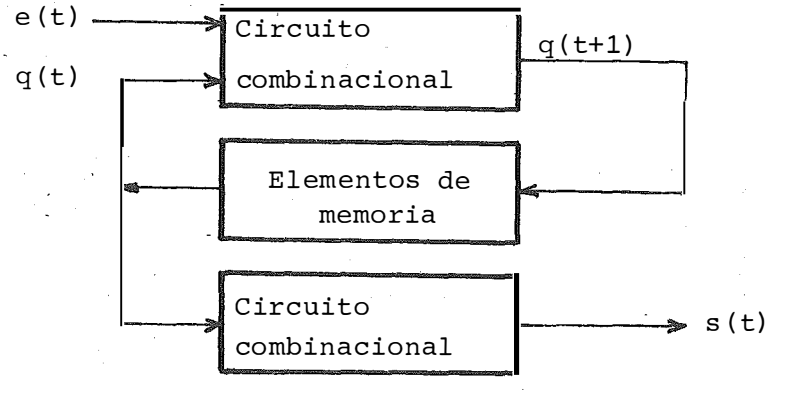
$$q(t+1) = f(e(t), q(t))$$

$$s(t) = g(e(t), q(t))$$

De ellas podemos deducir un primer modelo general para circuitos secuenciales (Fig. 3.5 a); consiste en disociar la parte combinacional, realizable mediante circuitos lógicos que calculan f y g, de la parte de memoria, que retarda en una unidad de tiempo q(t) y s(t). Este sería el modelo de Mealy, porque si existe una función de salida h: Q → S, tal que s (t) = = h(q(t)), podemos establecer el modelo de Moore (Fig. 3.5b).



a)



b)

Fig. 3.5.

Supongamos que E tiene 3 símbolos, a,b,c; como los circuitos combinacionales son binarios, tendremos que codificarlos, haciendo, por ejemplo, a = 00, b = 01, c = 10. Por tanto, la entrada "e" serán en realidad, 2 hilos de entrada binaria: uno para el primer dígito del código, y otro para el segundo. En general, si E tiene l elementos, tendremos n hilos, con n tal que $2^{n-1} < l \leq 2^n$. Del mismo modo, habremos de suponer que en la realización con tecnología binaria serán necesarios, en general, m hilos para la salida y p hilos para el estado. De acuerdo con esto, el modelo básico de Mealy en forma de diagrama de bloques será el de la Fig. 3.6, en donde ya se supone que por todos los hilos las señales son binarias, y de igual modo podríamos dibujar el modelo básico de Moore. Para prescindir en la escritura de la variable t hemos adoptado el convenio, que seguiremos en adelante, de llamar z_i a $q_i(t+1)$.

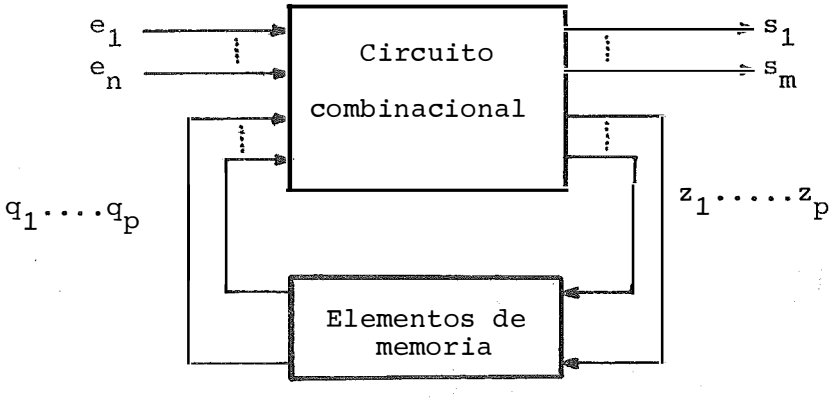


Fig. 3.6.

Con estas nuevas variables binarias la función f será:

$$z_i = f_i(e_1,\ldots, e_n,q_1,\ldots,q_p),\ i = 1,\ldots,m$$

y análogamente g y h. Si definimos los vectores columna $\underline{z},\underline{q},\underline{e},\underline{s}$ tendremos con notación vectorial:

$$\underline{z} = f(\underline{e},\underline{q})$$

$$\underline{s} = g(\underline{e},\underline{q})$$

$$\underline{s} = h(\underline{q})$$

4. TIPOS DE CIRCUITOS SECUENCIALES.

En el Apartado 2.3 hemos visto dos tipos de elementos con memoria: los no sincronizados, y los sincronizados con una señal de reloj, C. Por otra parte, las señales de entrada pueden ser de cuatro tipos: impulsionales síncronas o asíncronas y de nivel síncronas o asíncronas (Fig. 3.7). Si se utilizan unidades de memoria sincronizadas lo normal es que las señales de entrada sean síncronas, y así tenemos cuatro tipos de circuitos secuenciales:

- Tipo 1. Síncronos impulsionales. (Elementos de memoria sincronizados y entradas impulsionales síncronas.

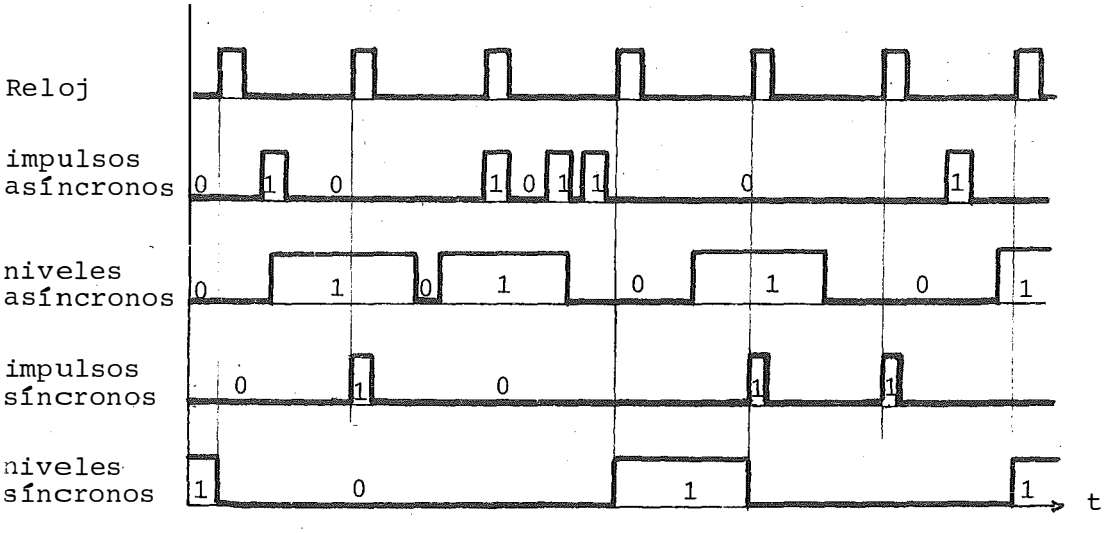


Fig. 3.7.

- Tipo 2. Síncronas de nivel (Igual que las anteriores, pero entradas de nivel).

- Tipo 3. Asíncronas impulsionales. (Elementos de memoria no sincronizados y entradas impulsionales asíncronas).

- Tipo 4. Asíncronas de nivel. (Como el tipo 3, con entradas de nivel).

Las salidas serán síncronas (impulsionales o de nivel) para el tipo 1, y lo mismo para el 2, asíncronas (impulsionales o de nivel) para el tipo 3 y necesariamente asíncronas de nivel para el tipo 4.

Dependiendo de las características particulares de cada aplicación se utiliza un tipo u otro de circuito secuencial. Aquí nos vamos a limitar para los ejemplos exclusivamente a los tipos 1 y 2

5. ANÁLISIS DE CIRCUITOS SECUENCIALES.

El análisis de un circuito secuencial consiste en obtener su salida para una determinada cadena de entrada, o bien obtener su representación ya sea formal, ya en diagramas o tablas. El problema es conceptualmente muy fácil, y vamos a ilustrarlo con un ejemplo sencillo.

Consideremos el circuito secuencial de la figura 3.8

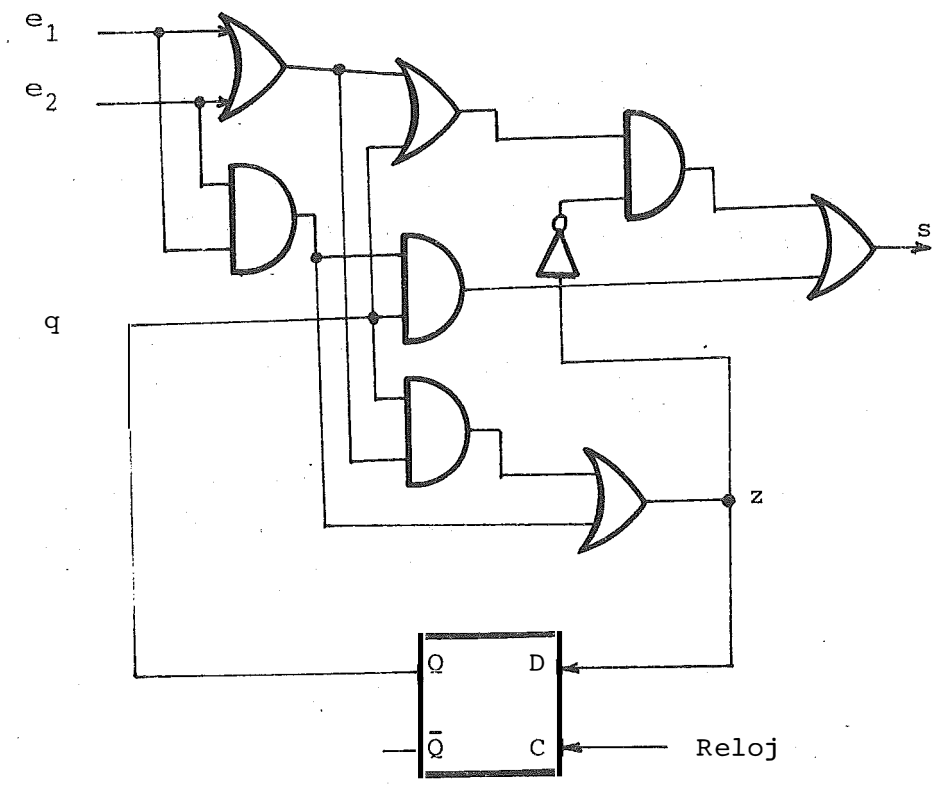


Fig. 3.8.

De la parte combinacional vemos que $s = e_1.e_2.q + \bar{z}.(e_1+e_2+q)$, y $z = e_1.e_2+q.(e_1+e_2)$, y de la parte de memoria, como es un biestable tipo D, $q = z.C+q.\bar{C}$, es decir, q mantiene su valor hasta que C pasa de 0 a 1, instante en el que q pasa a valer z. Supongamos que las señales de entrada son de nivel (sincronizadas con C), que el estado inicial del biestable es q = 0, y que introducimos las cadenas de entrada x_1 = 100110, x_2 = 101110. Durante el primer intervalo de tiempo - q=0, z=1.1+0.(1+1) = 1, s = 1.1.0 + 0.(1+1+0) = 0.Al llegar el siguiente impulso de reloj q pasa a valer 1 (pues z = 1), - z = 0.0 + 1.(0 + 0) = 0; s = 0.0.1 + 1.(0 + 0 + 1) = 1. Si - vamos anotando gráficamente estos resultados obtenemos el cronograma de la figura 3.9

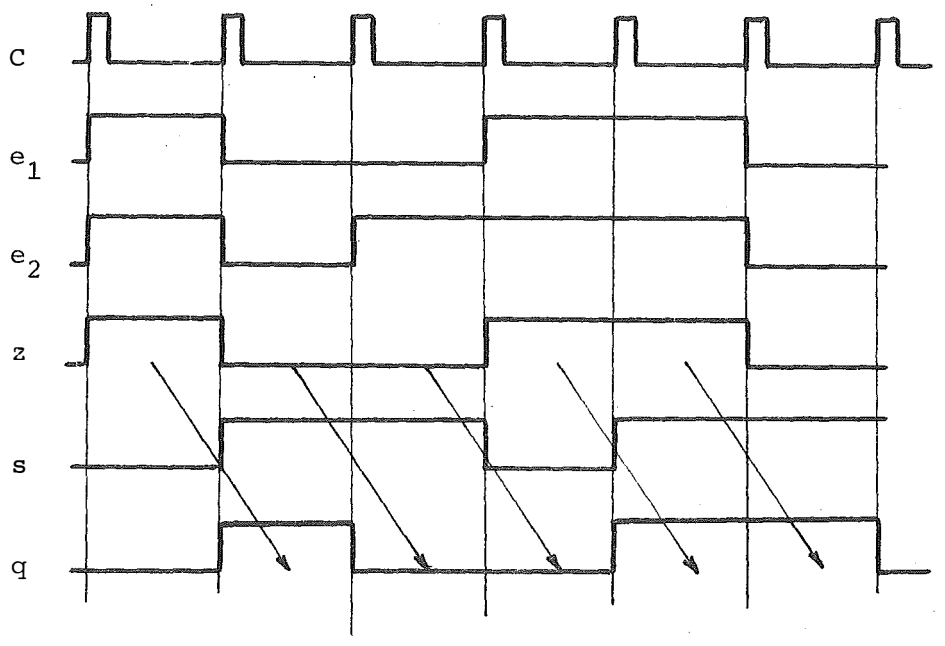


Fig. 3.9.

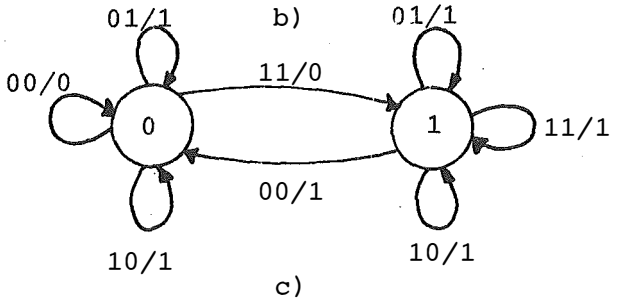
El cronograma es muy útil para ver gráficamente el comportamiento del circuito, pero sólo lo representa para una determinada cadena de entrada. Para tener una representación más general de este circuito podemos poner en forma de tabla todas las posibles combinaciones de e_1, e_2 y q y las resultantes para s y z. Esta tabla (Fig. 3.10 a) nos conduce inmediatamente a la tabla de transición (Fig. 3.10 b) y al diagrama de Moore (Fig. 3.10 c). Obsérvese que este diagrama es preci-

e_1	e_2	q	s	z
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

a)

q \ e	00	01	10	11
0	0/0	0/1	0/1	1/0
1	0/1	1/0	1/0	1/1

b)



c)

Fig. 3.10.

samente el del autómata sumador binario, es decir, el circuito de la figura 3.8 es la realización física de tal autómata. (La parte combinacional es una etapa de sumador).

6. SÍNTESIS DE CIRCUITOS SECUENCIALES.

6.1. Pasos de la síntesis

El diseño de un circuito secuencial se lleva a cabo siguiendo los siguientes pasos:

a) A partir de las especificaciones, deducir una tabla de transiciones y/o un diagrama de Moore.

b) Minimizar el AF.

c) Hacer una asignación de estados.

d) Escribir la tabla de transición en binario.

e) En función del tipo de biestable utilizado, deducir las tablas de excitación, que dan las entradas necesarias al biestable para cada transición.

f) De las tablas de excitación, obtener las ecuaciones lógicas de la parte combinacional.

g) Obtener las ecuaciones lógicas de $s = h(q)$.

Dada la diversidad de posibles especificaciones, no existe un método para el paso a).

Para el paso b) puede seguirse el algoritmo estudiado en el Apartado 5 del Capítulo 2.

El paso c) significa lo siguiente: Supongamos que el AF minimizado en el paso b) tiene N estados. Como los elementos de memoria de que disponemos son binarios, necesitaremos p elementos, con $2^{p-1} < N \leq 2^p$. La asignación de estados es una codificación arbitraria de los N estados con p dígitos binarios. El problema aquí es que según se haga una u otra codificación de las $(2^p!)/(2^p - N)!$ posibles, el circuito combinacional resultante puede ser más o menos complicado, y que no existe un método para saber cuál es la codificación óptima, aunque sí hay algunas reglas en las que no vamos a entrar, que el lector interesado puede estudiar en la bibliografía.

El paso e) consiste en obtener las tablas de verdad de la parte combinacional.

Para los pasos f) y g) aplicaremos las técnicas conocidas del Tema "Lógica".

6.2. Ejemplos.

6.2.1. Detector de paridad.

El diagrama de Moore es el de la figura 2.6. Si asignamos a los estados los valores $q_1 = 0$ y $q_2 = 1$, la tabla de transición en binario será:

q \ e	0	1
0/0	0	1
1/1	1	0

Escribamos la tabla de una forma lineal, es decir, - con una línea para cada combinación posible de e y q:

e	q	z
0	0	0
0	1	1
1	0	1
1	1	0

Supongamos ahora que vamos a utilizar un biestable tipo D (sólo hace falta uno, puesto que sólo hay dos estados). Deberemos anotar en cada línea la entrada necesaria al biestable para que tenga lugar la transición correspondiente. Por - ejemplo, en la primera línea, de q(t) = 0 se debe pasar a - - z = q(t+1) = 0. Para ello, D = 0; en la segunda línea D = 1, etc. Así, podemos poner la tabla:

e	q	D
0	0	0
0	1	1
1	0	1
1	1	0

que nos define D (entrada del biestable) en función de e(t) y q(t), y se llama tabla de excitaciones. De ella deducimos la función D:

$$D = \bar{e}.q + e.\bar{q} = e \oplus q,$$

y, como s = q, la realización del circuito será la de la figura 3.11 a, o bien la de la figura 3.11 b. En realidad, hemos tomado s = z, lo que nos permite obtener s con un intervalo de adelanto.

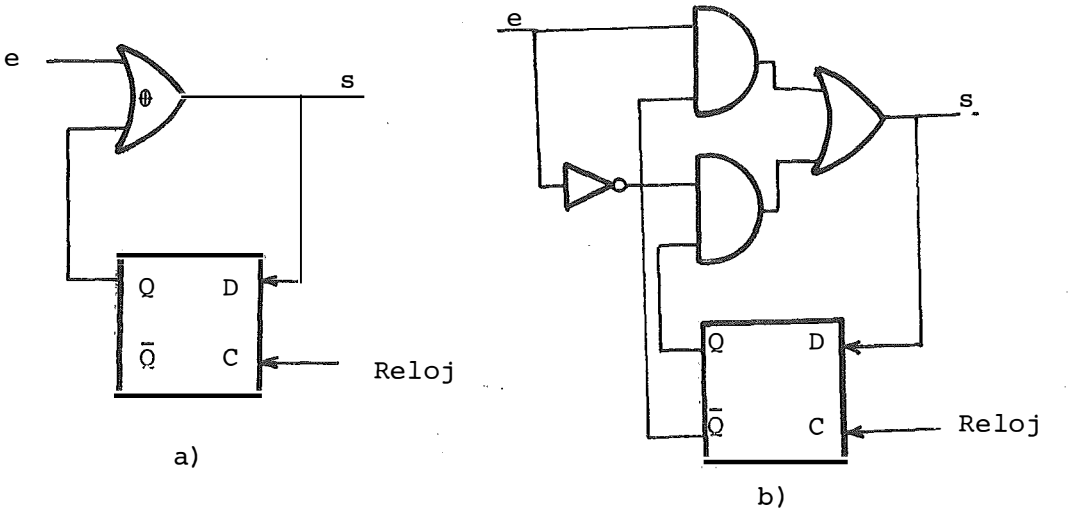


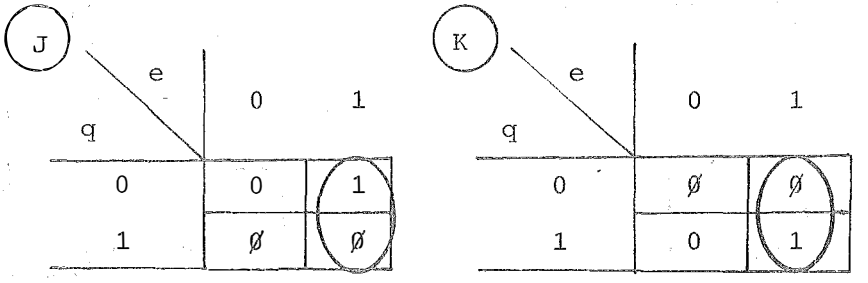
Fig. 3.11.

Obsérvese que, debido al modo de funcionamiento del biestable D, la tabla de excitaciones se obtiene, simplemente, sustituyendo "z" por "D", ya que la salida del biestable en t + 1 es justamente su entrada en t. No ocurre lo mismo con el JK. En efecto, si tenemos que realizar la transición de q = 0 a z = 0, las entradas J y K pueden ser ambas 0 o bien J = 0, K = 1 (puesta a cero); para la transición de q = 0 a Z = 1 podemos hacer J = 1, K = 0 (puesta a uno) o bien J = 1, K = 1 (complementación), etc. Así podemos escribir la tabla de excitaciones para este ejemplo:

e	q	Z	J	K
0	0	0	0	Ø
0	1	1	Ø	0
1	0	1	1	Ø
1	1	0	Ø	1

en la que hemos puesto "Ø" en los lugares donde es indiferente que sea "0" ó "1"; como sabemos del Tema "Lógica" esto nos

ayuda en la minimización del circuito. Las tablas de Karnaugh de J y K en función de e y q son:



de las que obtenemos

$$J = K = e,$$

por lo que la realización con JK será la de la figura 3.12

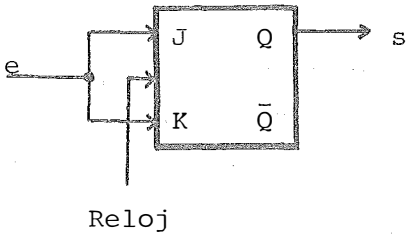


Fig. 3.12.

6.2.2. Reconocedor de 010

El diagrama de Moore del AF reconocedor de la cadena 010 es el de la figura 2.12. Como hay 4 estados necesitaremos 2 biestables. Hagamos la siguiente asignación de estados: $q_1 = 00$; $q_2 = 01$; $q_3 = 10$; $q_4 = 11$. Con ello podemos escribir la tabla de transiciones en binario, y, al mismo tiempo, para cada transición, las entradas de los biestables; si éstos van a ser JK obtenemos:

e	q	s	z	J_1	K_1	J_2	K_2
0	00	0	01	0	ø	1	ø
0	01	0	01	0	ø	ø	0
0	10	0	11	ø	0	1	ø
0	11	1	01	ø	1	ø	0
1	00	0	00	0	ø	0	ø
1	01	0	10	1	ø	ø	1
1	10	0	00	ø	1	0	ø
1	11	1	10	ø	1	ø	0

Las 4 últimas columnas con las dos primeras definen las tablas de excitaciones. Si llamamos Q_1 y Q_2 (salidas de los biestables) al primero y segundo dígito de q, minimizando por Karnaugh podemos hallar las ecuaciones lógicas de las entradas de los biestables en función de e, Q_1 y Q_2; el resultado es:

$$J_1 = e.Q_2;\ K_1 = e + Q_2;\ J_2 = \bar{e};\ K_2 = \bar{e}.\bar{Q}_1$$

Por otra parte, de las 3 primeras columnas se tendrá la función de salida, es decir, s en función de e, Q_1, Q_2. Minimizando resulta:

$$s = Q_1.Q_2$$

Con lo cual podemos trazar el circuito de la figura 3.13

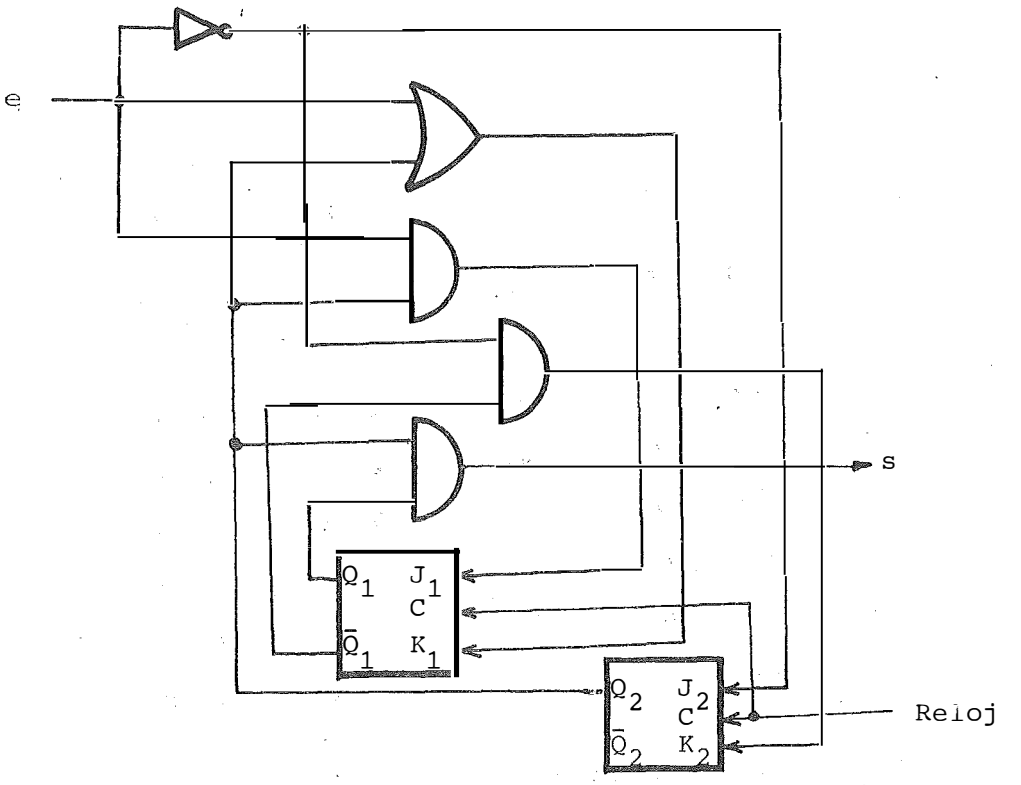


Fig. 3.13.

6.2.3. Contador BCD módulo 10.

Partimos del diagrama de Moore de la figura 2.17. Como hay 10 estados necesitaremos 4 biestables ($2^3 < 10 < 2^4$), que supondremos del tipo JK. Codifiquemos los estados con los mismos dígitos que las salidas asociadas, es decir:

$$q_0 = s_0 = 0000$$
$$q_1 = s_1 = 0001$$
$$q_2 = s_2 = 0010$$
$$\vdots$$
$$q_9 = s_9 = 1001$$

De esta manera se simplifica al máximo el circuito que realiza la función de salida, puesto que será s = q.

Escribimos a continuación la tabla de transición junto

con la de excitaciones

e	Q_3	Q_2	Q_1	Q_0	Z	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0000	0	ø	0	ø	0	ø	0	ø
0	0	0	0	1	0001	0	ø	0	ø	0	ø	ø	0
0	0	0	1	0	0010	0	ø	0	ø	ø	0	0	ø
0	0	0	1	1	0011	0	ø	0	ø	ø	0	ø	0
0	0	1	0	0	0100	0	ø	ø	0	0	ø	0	ø
0	0	1	0	1	0101	0	ø	ø	0	0	ø	ø	0
0	0	1	1	0	0110	0	ø	ø	0	ø	0	0	ø
0	0	1	1	1	0111	0	ø	ø	0	ø	0	ø	0
0	1	0	0	0	1000	ø	0	0	ø	0	ø	0	ø
0	1	0	0	1	1001	ø	0	0	ø	0	ø	ø	0
1	0	0	0	0	0001	0	ø	0	ø	0	ø	1	ø
1	0	0	0	1	0010	0	ø	0	ø	1	ø	ø	1
1	0	0	1	0	0011	0	ø	0	ø	ø	0	1	ø
1	0	0	1	1	0100	0	ø	1	ø	ø	1	ø	1
1	0	1	0	0	0101	0	ø	ø	0	0	ø	1	ø
1	0	1	0	1	0110	0	ø	ø	0	1	ø	ø	1
1	0	1	1	0	0111	0	ø	ø	0	ø	0	1	ø
1	0	1	1	1	1000	1	ø	ø	1	ø	1	ø	1
1	1	0	0	0	1001	ø	0	0	ø	0	ø	1	ø
1	1	0	0	1	0000	ø	1	0	ø	0	ø	ø	1

Tenemos así en forma de tablas de verdad J_i y K_i en función de e, Q_1, Q_2, Q_3, Q_4. Obsérvese que no están todas -

las combinaciones de variables binarias de estado, debido a - que sólo tenemos 10 estados de los 16 posibles con 4 varia- - bles. Para tales combinaciones podemos tomar cualquier valor (Ø) en las J y K.

La tabla de Karnaugh de J_3 será:

Q_1Q_0 \ Q_3Q_2	00	01	11	10
00	0	0	Ø	Ø
01	0	0	Ø	Ø
11	0	0	Ø	Ø
10	0	0	Ø	Ø

e = 0

Q_1Q_0 \ Q_3Q_2	00	01	11	10
00	0	0	Ø	Ø
01	0	0	Ø	Ø
11	0	1	Ø	Ø
10	0	0	Ø	Ø

e = 1

Y de ella deducimos:

$$J_3 = e.Q_2.Q_1.Q_0$$

Y, análogamente,

$$K_3 = e.Q_0,$$

$$J_2 = e.Q_1.Q_0; \qquad K_2 = e.Q_1.Q_0$$

$$J_1 = e.\bar{Q}_3.Q_0; \qquad K_1 = e.Q_0$$

$$J_0 = e \quad ; \qquad K_0 = e$$

Resulta así el circuito de la figura 3.14

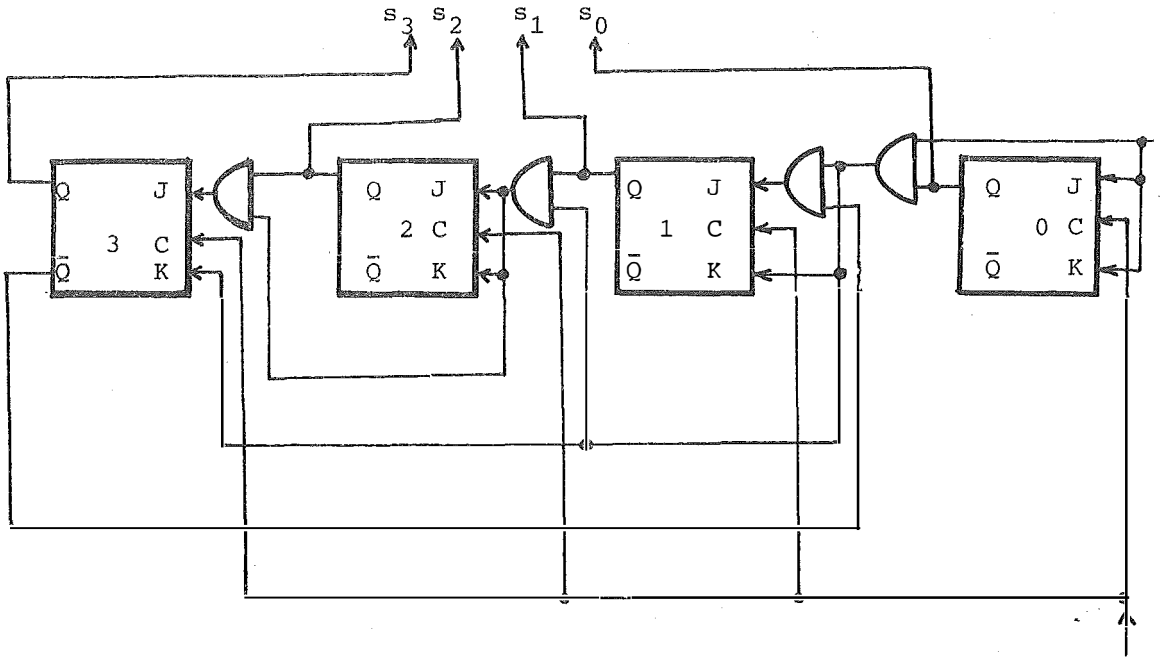


Fig. 3.14.

7. RESUMEN.

Hemos estudiado algunos de los elementos utilizados - como memorias en la realización física de los AF (circuitos - secuenciales). Conociendo el funcionamiento de los elementos que lo componen, el análisis de un circuito secuencial se puede llevar a cabo deduciendo paso a paso el cronograma correspondiente a una determinada cadena, y también se puede obtener el diagrama de Moore, que tendrá, si p es el número de - elementos binarios de memoria, 2^p estados.

Hemos visto el procedimiento general para la síntesis de circuitos secuenciales, ilustrándolo con algunos ejemplos. No hemos abordado, por su mayor complejidad, el diseño de máquinas incompletamente especificadas ni el problema de la - - asignación de estados.

8. Notas histórica y bibliográfica.

La tecnología de realización de circuitos digitales ha evolucionado muy rápidamente en los últimos años, pero las técnicas básicas de diseño, independientes de esa tecnología, son las mismas de los años 60. Por eso, libros de esa época, como el de BARTEE et al. (1.962) constituyen aún una buena referencia.

El alumno tendrá ocasión de ampliar los temas tratados aquí muy superficialmente en la asignatura de Electrónica de 4° curso, y en la de Sistemas digitales, de 5°. Para el que ya esté interesado en profundizar en este campo podemos recomendar los libros de MUÑOZ (1.977) (texto en la asignatura "Electrónica"), MANDADO (1.977) y NAGLE et al. (1.975).

9. Ejercicios.

1. Diseñar el sumador binario serie utilizando un biestable JK.

2. En el ejercicio 10 del Capítulo 2, supóngase que los tres posibles símbolos de entrada se codifican en binario:1=01; 2 = 10; 3 = 11 (es decir, habrá dos hilos de entrada). Diseñar el circuito secuencial correspondiente.

3. El código "exceso de 3" es un código BCD en el que la codificación de cada dígito natural se obtiene sumando $11)_2$ a la correspondiente codificación en BCD natural, es decir, 0 se codifica "0011"; 1, "0100"; 2, "0101";....;9,"1100" (este código permite una complementación directa a 9 y simplifica la suma decimal). Diseñar un contador módulo 10 en

exceso de 3 utilizando biestables JK.

4. Diseñar contadores módulo 10 en BCD natural y en exceso de 3 con biestables tipo D.

5. Diseñar circuitos con biestables JK y D para un autómata retardador de 2 intervalos (s(t) = e(t-2)).

6. Diseñar un circuito que gobierne el funcionamiento de tres luces (verde, roja y amarilla) reguladoras de tráfico. La luz verde deberá permanecer encendida durante 40 seg., con la roja apagada; al cabo de este tiempo la situación cambiará (verde apagada, roja encendida) durante otros 40 seg., y así sucesivamente. La luz amarilla sólo se encenderá (simultáneamente con la verde) durante los diez segundos que preceden al encendido de la roja. Para ello, el circuito tendrá tres salidas binarias (V,R,A) y una sola entrada consistente en impulsos de período 10 seg.

7. Diseñar un circuito secuencial que simule la situación descrita para el "castillo encantado" (Apartado 1.4.3 del Capítulo 2).

CAPITULO 4.

AUTOMATAS RECONOCEDORES Y LENGUAJES REGULARES

1. RECONOCEDOR FINITO

1.1. Definición

Hemos definido en el Capítulo 2 un lenguaje, L, sobre un alfabeto, E, como un subconjunto cualquiera de E^*. En este Capítulo vamos a ver que ciertos lenguajes pueden asociarse con autómatas finitos que sirven como reconocedores de las cadenas pertenecientes a tales lenguajes.

Un reconocedor finito de un lenguaje L es un AF que sólo acepta las cadenas de dicho lenguaje, en el sentido de que, inicializado en un estado predeterminado, q_1, si se le introduce una cadena de entrada $x_1 \in L$, da un símbolo final de salida que corresponde a "aceptación" (por ejemplo, s = 1), mientras que para $x_1 \notin L$ produce una salida de "no aceptación" (por ejemplo, s = 0). En adelante supondremos siempre que hablemos de reconocedores que se trata de máquinas de Moore. Podemos entonces hacer abstracción del alfabeto de salida y de la función g, y considerar el subconjunto de estados $F \subseteq Q$ que producen la salida de aceptación, a los que llamaremos estados finales. De acuerdo con esto, daremos la siguiente definición formal:

Un reconocedor finito es una quíntupla

$$R = \langle E, Q, f, q_1, F \rangle$$

onde:

E es un conjunto finito (alfabeto de entrada)

Q es un conjunto finito (conjunto de estados)

f es una función f: E x Q → Q (función de transición)

$q_1 \in Q$ es un estado designado como estado inicial

$F \subset Q$ es un conjunto de estados designados como estados finales.

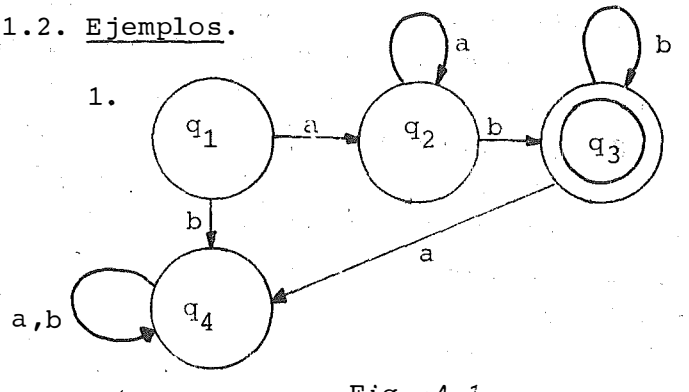
Llamaremos L (R) al conjunto de cadenas aceptadas por R, es decir

$$L(R) = \{x \in E^* \mid f(x,q_1) \in F\}$$

Seguiremos el convenio de representar en los diagramas de Moore los estados de aceptación con un círculo doble, y en las tablas de transición, encerrados en un círculo.

1.2. Ejemplos.

1.



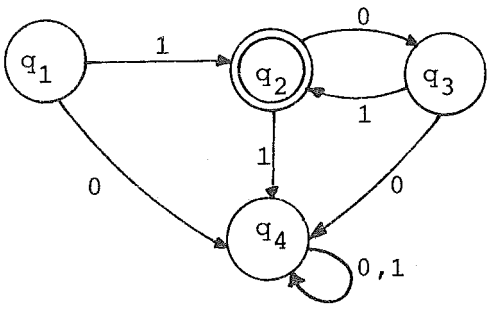
	a	b
q_1	q_2	q_4
q_2	q_2	q_3
(q_3)	q_4	q_3
q_4	q_4	q_4

Fig. 4.1.

$$L(R_1) = \{ab, aab, \ldots abb, abbb, \ldots aabb, \ldots\} =$$

$$= \{a^n b^m \mid n \geq 1,\ m \geq 1\}$$

2.

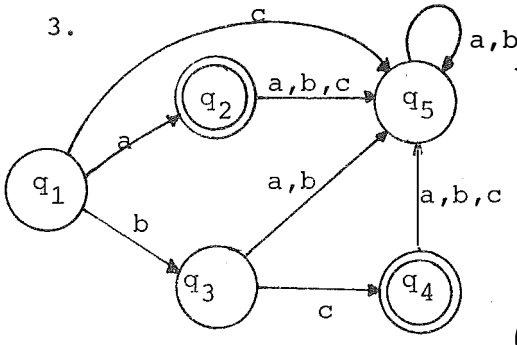


	0	1
q_1	q_4	q_2
(q_2)	q_3	q_4
q_3	q_4	q_2
q_4	q_4	q_4

Fig. 4.2.

$L(R_2) = \{1, 101, 10101, \ldots\} = \{1(01)^n \mid n \geq 0\}$

3.

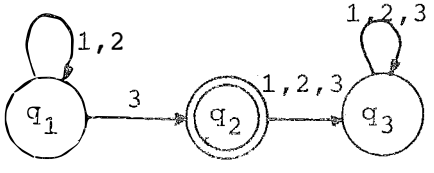


a,b,c	a	b	c
q_1	q_2	q_3	q_5
(q_2)	q_5	q_5	q_5
q_3	q_5	q_5	q_4
(q_4)	q_5	q_5	q_5
q_5	q_5	q_5	q_5

Fig. 4.3.

$L(R_3) = \{a, bc\}$

4.



	1	2	3
q_1	q_1	q_1	q_2
(q_2)	q_3	q_3	q_3
q_3	q_3	q_3	q_3

Fig. 4.4.

$L(R_4) = \{3, 13, 113, 1113, \ldots 123, 1223, \ldots, 213, 2113, \ldots\} =$

$= \{1^{n_1} 2^{n_2} 1^{n_3} 2^{n_4} \ldots 3 \mid n_1, n_2 \ldots \geq 0\}$

2. LENGUAJES ACEPTADOS POR RECONOCEDORES FINITOS.

2.1. Planteamiento del problema.

Podemos preguntarnos si, dado un lenguaje cualquiera, $L \subset E^*$, podemos siempre encontrar un reconocedor finito, R, - tal que L (R) = L. Observemos que esa pregunta ya la habíamos dejado planteada, en términos más generales, en el Capítulo 2, cuando, al final del Apartado 3.1, decíamos: dada una máquina ¿podemos encontrar su circuito?.

Baste un ejemplo para demostrar que existen lenguajes a los que no corresponde ningún reconocedor finito. Consideremos $E = \{0,1\}$, y sea $L = \{1^{n^2} | n \geq 1\}$. Supongamos que existe - un R, con p estados, tal que L (R) = L. Para cadenas del tipo $x = 1^i$ tendremos que $f(1^i,q_1) \in F$ si i es un cuadrado perfecto. Para las p+1 cadenas $x_0 = 1^0$, $x_1 = 1^1 \ldots . x_p = 1^p$ tendremos como estados resultantes $f(x_o,q_1)$, $f(x_1,q_1)$, ..., $f(x_p,q_1)$, pero como el reconocedor sólo tiene p estados, al menos dos - de estos estados resultantes deberán ser el mismo: $f(1^i,q_1) = f(1^j,q_1)$, con $j - 1 \leq p$. Entonces, si el reconocedor acepta 1^{n^2} también deberá aceptar $1^{n^2 + (j-i)}$, ya que $f(1^{n^2},q_1) = f(1^i 1^{n^2 - i}, q_1) = f\,[1^{n^2-i}, f(1^i,q_1)] = f\,[1^{n^2-i}, f(1^j,q_1)] = f(1^j 1^{n^2-i}, q_1) = f(1^{n^2+(j-i)}, q_1)$. Ahora bien, siempre podemos tomar n suficientemente grande como para que $(n+1)^2 - n^2 > p$, y, por tanto, $(n+1)^2 - n^2 > (j-i)$, con lo que $n^2 + (j-i) < (n+1)^2$ no será un cuadrado perfecto, y el reconocedor responderá incorrectamente al aceptar $1^{n^2+(j-i)}$.

Vemos pues que *los lenguajes aceptados por algún reconocedor finito son un subconjunto de todos los lenguajes posibles sobre* E^*.

Ahora podemos preguntarnos: *¿existe alguna propiedad que caracterice a los lenguajes que son aceptados por un reconocedor finito?*. Si tal propiedad existe parece lógico pensar que pueda deducirse de la capacidad del reconocedor para responder de distinto modo a diferentes cadenas de entrada.

2.2. Relación equirespuesta de un reconocedor finito.

En el Capítulo 2 definimos de una manera general el comportamiento de entrada-estados de un AF como un homomorfismo $K: \langle E^*, \rangle \rightarrow \langle Q^Q, \circ \rangle$, tal que a cada $x \in E^*$ corresponde una transformación entre estados, $K(x): Q \rightarrow Q$, y esto nos permitió definir una relación equirespuesta en E^* tal que $x \cong y$ si y sólo si $K(x) = K(y)$, es decir, las cadenas x e y estarán en relación si y sólo si producen la misma transformación entre estados: $(x \cong y) \leftrightarrow f(x,q_i) = f(y,q_i)$, $\forall q_i \in Q$. Veíamos que esta relación es una relación de congruencia en $\langle E^*, \rangle$ y particiona E^* en un número finito de clases de equivalencia (es decir, es de índice finito, menor o igual a n^n, siendo n el número de estados).

En el caso del reconocedor finito, el estado inicial, q_1, es fijo, y el comportamiento de entrada-estados será más bien una función $K_R: E^* \rightarrow Q$ que aplica a cada $x \in E^*$ un $q \in Q$ de tal manera que $K_R(x) = f(x,q_1)$. Podemos igualmente definir una relación equirespuesta, $\cong_R$, en E^*:

$$x \cong_R y \text{ si y sólo si } f(x,q_1) = f(y,q_1), \ \forall x, y \in E^*$$

Esta relación, como es fácil ver, es una relación de equivalencia. Además, $(x \cong y) \rightarrow (x \cong_R y)$, pero no a la inversa (se dice que $\cong$ refina a $\cong_R$), por lo que en la partición de E^* inducida por $\cong_R$ habrá un número igual o inferior de clases de equi

valencia que en la inducida por $\cong$.

Si bien $\cong$ es una relación de congruencia en $\langle E^*, \rangle, \cong_R$ sólo es una relación de congruencia derecha. Esto quiere decir que $(x \cong_R y) \rightarrow (xz \cong_R yz)$, pero en general, no es cierto que $(zx \cong_R zy)$. En efecto, si llamamos $q_i = f(x,q_1) = f(y,q_1)$ y $q_j = f(z,q_i)$, tendremos: $f(xz,q_1) = f(z,f(x,q_1)) = f(z,q_i) = q_j$ y análogamente $f(yz,q_1) = q_j$; sin embargo $f(zx,q_1) = f[x,f(z,q_1)]$ que en general será diferente de $f[y,f(z,q_1)]$

Así pues, a cada reconocedor finito corresponde una partición de E^* en clases de equivalencia tal que si dos cadenas están en la misma clase ambas cadenas conducen al mismo estado. Además, esta relación de equivalencia es de índice finito (menor o igual que n^n, siendo n el número de estados del reconocedor) y es una relación de congruencia derecha en E^*. Veamos cómo pueden aplicarse estas conclusiones para caracterizar a los lenguajes aceptados por reconocedores finitos.

2.3. Condición para que un lenguaje sea aceptado por un reconocedor finito.

Dado un lenguaje $L \subset E^*$, definimos la relación de congruencia derecha inducida por L, $\cong_L$, así:

$$(x \cong_L y) \leftrightarrow (xz \in L \leftrightarrow yz \in L) \quad \forall x,y,z \in E^*$$

Es evidente que se trata de una relación de equivalencia. Para ver que también es una congruencia derecha basta suponer que $z = z_1 z_2$, con lo que

$$(x \cong_L y) \leftrightarrow (xz_1z_2 \in L \leftrightarrow yz_1z_2 \in L) \leftrightarrow (xz_1 \cong_L yz_1) \quad \forall x,y,z_1,z_2 \in E^*$$

Vamos a demostrar ahora la principal conclusión de este Apartado 2:

$L \subset E^*$ es un lenguaje aceptado por un reconocedor finito si y sólo si la relación de congruencia derecha inducida por L tiene índice finito.

Veamos primero que si L = L(R) entonces $\cong_L$ tiene índice finito:

a) Por definición de $\cong_R$, $(x \cong_R y) \leftrightarrow (f(x,q_1) = f(y,q_1))$

b) Por ser $\cong_R$ una congruencia derecha,

$$(x \cong_R y) \rightarrow [f(xz,q_1) = f(yz,q_1)]$$

c) Es claro que

$$[f(xz,q_1) = f(yz,q_1)] \rightarrow [(f(xz,q_1) \in F) \leftrightarrow (f(yz,q_1) \in F)]$$

d) y también que

$$[(f(xz,q_1) \in F) \leftrightarrow (f(yz,q_1) \in F)] \leftrightarrow [(xz \in L) \leftrightarrow (yz \in L)]$$

e) Por definición de $\cong_L$,

$$[(xz \in L) \leftrightarrow (yz \in L)] \leftrightarrow (x =_L y)$$

f) El razonamiento constituido por las premisas b), c), d) y e) nos lleva a la conclusión de que

$$(x \cong_R y) \rightarrow (x \cong_L y)$$

($\cong_R$ refina a $\cong_L$). Por consiguiente, el índice (número

de clases de equivalencia) de la partición inducida en E^* por $\cong_L$ es igual o inferior al de la partición inducida por $\cong_R$, y como ésta es de índice finito, $\cong_L$ también lo será

A la inversa, supongamos que L es un lenguaje que induce una relación de congruencia derecha en E^* que es de índice finito. Vamos a construir un reconocedor finito, R_L, tal que $L(R_L) = L$. Llamemos $[x]$ a la clase de equivalencia de $E^*/\cong_L$ que contiene a $x \in E^*$. Entonces definimos

$$R_L = \langle E, Q_L, f_L, q_1, F_L \rangle,$$

donde

$Q_L = E^*/\cong_L = \{[x]\}$ (finito, puesto que $\cong_L$ es de índice finito)

$f_L(y, [x]) = [xy] \quad \forall y \in E^*$

$q_1 = [\Lambda]$

$F_L = \{[x] \mid x \in L\}$

(F_L está bien definida, pues si $x_1 \in L$ y $x_2 \in L$, según la definición de $\cong_L$, haciendo $z = \Lambda$, vemos que $x_1 \cong_L x_2$, es decir, $[x_1] = [x_2]$)

El lenguaje aceptado por este reconocedor será:

$$L(R_L) = \{y \in E^* \mid f_L(y, q_1) \in F_L\} = \{y \mid f_L(y, [\Lambda]) \in F_L\} =$$

$$= \{y \mid [y] \in F_L\} = \{y \mid y \in L\} = L$$

Además, R_L está en forma mínima. En efecto, si $q_{L_1} = [x_1]$ fuera equivalente a $q_{L_2} = [x_2]$, esto querría decir que $(f_L(y,[x_1]) \in L) \leftrightarrow (f_L(y,[x_2]) \in L)$, $\forall y \in E^*$, y por la definición de f_L, $([x_1y] \in L) \leftrightarrow ([x_2y] \in L)$, $\forall y \in E^*$, es decir, $x_1 \cong_L x_2$; x_1 y x_2 están en la misma clase de equivalencia, por lo que $q_{L_1} = q_{L_2}$.

3. CONJUNTOS REGULARES Y EXPRESIONES REGULARES.

3.1. Los problemas de análisis y de síntesis.

Acabamos de ver una condición necesaria y suficiente para que un lenguaje sea aceptado por un reconocedor finito. El problema de análisis consiste en deducir el lenguaje asociado a un determinado reconocedor, y el de síntesis en encontrar un reconocedor cuyo lenguaje sea un lenguaje dado. Ambos problemas los tenemos en teoría resueltos. En efecto, para el análisis, basta enumerar las cadenas que son aceptadas, como vimos en los ejemplos de 1.2.; para la síntesis, hay que encontrar las clases de equivalencia de la relación de congruencia derecha inducida por L y construir el reconocedor como se ha indicado más arriba. El inconveniente está en que, al poder ser L un conjunto infinito, no es fácil trabajar con él, y no siempre podemos representarlo de una manera condensada, como hacíamos en los ejemplos de 1.2.; además, salvo en el caso de que L sea finito (como en el ejemplo 3), no tenemos un algoritmo para encontrar las clases de equivalencia inducidas por L.

En este Apartado vamos a exponer una herramienta, las

expresiones regulares, especialmente introducida para trabajar con los lenguajes aceptados por reconocedores finitos, y que nos permitirá llegar a algoritmos para resolver los problemas de análisis y de síntesis.

3.2. Conjuntos regulares.

En primer lugar vamos a definir tres operaciones en el conjunto $\{L_1, L_2 \ldots\}$ de subconjuntos de E^* (lenguajes sobre E):

a) Unión:

$$L_1 \cup L_2 = \{x | x \in L_1 \vee x \in L_2\}$$

b) Concatenación:

$$L_1 L_2 = \{x_1 x_2 | x_1 \in L_1 \wedge x_2 \in L_2\}$$

c) Cierre u operación estrella:

$$L^* = \{\Lambda\} \cup \{L\} \cup \{LL\} \cup \{LLL\} \ldots = \bigcup_{n=0}^{\infty} L^n$$

Decimos que un subconjunto de E^*, $L_R \subset E^*$, es regular si y sólo si

a) L_R es un subconjunto finito de E^* (puede ser $L = \emptyset$), o bien,

b) L_R puede obtenerse a partir de subconjuntos finitos de E^* mediante un número finito de operaciones de unión, concatenación y cierre.

3.3. Expresiones regulares

Las expresiones regulares se introducen para describir los conjuntos regulares, y como éstos son lenguajes, las expresiones regulares serán metalenguajes.

Seguiremos el convenio de designar por $|\alpha|$ al conjunto descrito por la expresión regular α.

Definimos ahora el conjunto de expresiones regulares sobre un alfabeto $E = \{e_1, \ldots, e_n\}$ y las operaciones suma, concatenación y cierre de la siguiente manera recursiva:

a) Λ, $\emptyset$ y e_i $(i=1,\ldots, n)$ son expresiones regulares tales que $|\Lambda| = \{\Lambda\}$; $|\emptyset| = \emptyset$ y $|e_i| = \{e_i\}$ $(i=1,\ldots,n)$;

b) si α y β son expresiones regulares, $\alpha + \beta$ es una expresión tal que $|\alpha + \beta| = |\alpha| \cup |\beta|$;

c) si α y β son expresiones regulares, $\alpha\beta$ es una expresión regular tal que $|\alpha\beta| = |\alpha|\,|\beta|$;

d) si α es una expresión regular, α^* es una expresión regular tal que $|\alpha^*| = |\alpha|^*$.

Como estas tres operaciones corresponden a las utilizadas para definir los conjuntos regulares, a todo conjunto corresponderá al menos una expresión regular.

Veamos algunos ejemplos.

	E	Expresión regular, α	Conjunto regular, $\|\alpha\|$
1.	{a,b}	aa^*bb^*	Conjunto de todas las cadenas de E^* constituídas por "a" seguido de "a" cualquier número de veces (o ninguna), seguido de "b" y seguido de "b" cualquier número de veces (o ninguna).
2.	{0,1}	$1(01)^*$	Conjunto de cadenas que empiezan por "1" y sigue (01) cualquier número de veces (o ninguna).
3.	{a,b,c }	a+bc	{a, bc}
4.	{1,2,3}	$(1+2)^*3$	Conjunto de cadenas formadas con los símbolos 1 y 2 sucediéndose cualquier número de veces (y en cualquier orden), y siempre terminando la cadena con el símbolo 3.
5.	$\{e_1,e_2,...,e_n\}$	$(e_1+e_2+...+e_n)^*$	E^*
6.	{0,1}	$(01)^*$	Conjunto formado por Λ y todas las cadenas constituidas por la cadena 01 repetida cualquier número de veces.
7.	{0,1}	0^*10^*	Conjunto de todas las cadenas que tienen un "1" (y sólo uno)

Obsérvese que los cuatro primeros ejemplos corresponden exactamente a los cuatro ejemplos de 1.2.

Dos expresiones regulares son iguales si designan al mismo conjunto regular:

$$(\alpha = \beta) \leftrightarrow |\alpha| = |\beta|$$

Teniendo esto presente, es fácil demostrar las siguientes propiedades de las expresiones regulares:

1. Asociatividad de la concatenación: $\alpha(\beta\gamma) = (\alpha\beta)_\gamma$

2. Distributividad de la suma: $\alpha\beta + \alpha\gamma = \alpha(\beta + \gamma)$;

   $\beta\alpha + \gamma\alpha = (\beta + \gamma)\ \alpha$

3. $\emptyset$ es elemento neutro para la suma: $\alpha + \emptyset = \emptyset + \alpha = \alpha$

4. $\emptyset$ es un cero para la concatenación: $\alpha\emptyset = \emptyset\alpha = \emptyset$

5. Λ es elemento neutro para la concatenación: $\alpha\Lambda = \Lambda\alpha = \alpha$

6. Propiedades de la operación cierre:

   a) $(\alpha + \beta)^* = (\alpha^* + \beta^*)^* = (\alpha^*\beta^*)^*$

   b) $(\alpha + \Lambda)^* = \alpha^* + \Lambda = \alpha^*$

   c) $\alpha\alpha^* + \Lambda = \alpha^*$

   d) $\Lambda^* = \emptyset^* = \Lambda$

Por ejemplo, para demostrar que $(\alpha + \beta)^* = (\alpha^*\beta^*)^*$ -

tendremos en cuenta que:

$$|(\alpha + \beta)^*| = \{\Lambda\} \cup |\alpha + \beta| \cup |\alpha + \beta|^2 \cup \cdots =$$

$$= \{\Lambda\} \cup |\alpha| \cup |\beta| \cup |\alpha|^2 \cup |\beta|^2 \cup |\alpha||\beta| \cup |\beta||\alpha| \cup \cdots$$

y por otra parte

$$|(\alpha^*\beta^*)^*| = \{\Lambda\} \cup |\alpha^*\beta^*| \cup |\alpha^*\beta^*|^2 \cup \cdots =$$

$$= \{\Lambda\} \cup |\alpha^*||\beta^*| \cup |\alpha^*||\beta^*| \cup |\alpha^*||\beta^*| \cup \cdots =$$

$$= \{\Lambda\} \cup |\alpha| \cup |\beta| \cup |\alpha|^2 \cup |\beta|^2 \cup |\alpha||\beta| \cup |\beta||\alpha| \cup \cdots$$

Pasemos ahora a ver que *todos los lenguajes aceptados por reconocedores finitos son conjuntos regulares y que todo conjunto regular es un lenguaje aceptado por un reconocedor - finito, lo que nos va a permitir resolver los problemas de - análisis y de síntesis, respectivamente.*

4. RESOLUCIÓN DE LOS PROBLEMAS DE ANÁLISIS Y DE SÍNTESIS DE UN RECONOCEDOR FINITO.

4.1. Análisis

4.1.1. Teorema de análisis

Todo lenguaje aceptado por un reconocedor finito es - un conjunto regular.

Supongamos que el reconocedor finito tiene n estados, $Q = \{q_1, \ldots, q_n\}$, con estado inicial q_1 y con estados finales $F = \{q_{f_1}, q_{f_2}, \ldots q_{f_r}\}$ $(n \geq r \geq 0)$. El lenguaje aceptado es:

$$L(R) = \{x \mid f(x,q_1) \in F\} = \bigcup \{R_{1j} \mid q_j \in F\},$$

con $R_{ij} = \{x \mid f(x,q_i) = q_j\}$ (conjunto de cadenas que llevan del estado q_i al estado q_j). Basta entonces demostrar que los R_{ij} son conjuntos regulares.

Vamos a definir R_{ij}^k $(0 \leq k \leq n)$ como el conjunto de cadenas que llevan de q_i a q_j sin pasar por ningún q_l tal que $l > k$. En particular, R_{ij}^0 serán las cadenas que llevan de q_i a q_j sin pasar por ningún otro estado, por lo que son símbo-- los, es decir, es un subconjunto de $E \cup \{\Lambda\}$ y, por tanto, es - regular. Supongamos que R_{ij}^{k-1} es regular para todo i y j y $k \geq 1$ y demostremos que entonces R_{ij}^k es regular. En efecto, basta - comprobar que

$$R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1} \, (R_{kk}^{k-1})^* \, R_{kj}^{k-1}$$

Entonces, R_{ij}^k es regular para todo k, y, en particu-- lar, $R_{ij} = R_{ij}^n$ es regular.

4.1.2. Algoritmo de análisis.

Como corolario del teorema anterior se desprende un - procedimiento para obtener la expresión regular del lenguaje aceptado por un determinado reconocedor. En efecto, llamamos

α_{ij}^k a la expresión regular que designa al conjunto R_{ij}^k: $|\alpha_{ij}^k| = R_{ij}^k$; entonces, si q_1 es el estado inicial, la expresión regular de L(R) será

$$\begin{cases} \alpha_{1f_1} + \alpha_{1f_2} + \ldots\ldots + \alpha_{1f_r} & \text{si } r > 0 \\ \emptyset \text{ si } r = 0 \end{cases}$$

Los α_{ij} se calcularán recursivamente teniendo en cuenta los conjuntos a los que representan:

$$|\alpha_{ij}^0| = \{e \in E \cup \{\Lambda\} \mid f(e,q_i) = q_j\}$$

$$\alpha_{ij}^k = \alpha_{ij}^{k-1} + \alpha_{ik}^{k-1} (\alpha_{kk}^{k-1})^* \alpha_{kj}^{k-1} \quad , \quad 0 < k \leq n$$

$$\alpha_{ij} = \alpha_{ij}^n$$

4.1.3. Ejemplos

Vamos a tomar los mismos cuatro ejemplos de 1.2: partimos de los diagramas, y debemos llegar a la expresiones regulares que ya conocemos.

1. Sólo hay un estado final, q_3, luego la expresión regular será

$$\alpha_{13} = \alpha_{13}^4$$

$$\alpha_{13}^4 = \alpha_{13}^3 + \alpha_{14}^3 (\alpha_{44}^3)^* \alpha_{43}^3$$

Pero $\alpha_{43}^{3} = \emptyset$, como se ve directamente en el diagrama de Moore, por lo que

$$\alpha_{13}^{4} = \alpha_{13}^{3} = \alpha_{13}^{2} + \alpha_{13}^{2} \, (\alpha_{33}^{2})^{*} \, \alpha_{33}^{2}$$

Calculemos pues α_{13}^{2} y α_{33}^{2} :

$$\alpha_{13}^{2} = \alpha_{13}^{1} + \alpha_{12}^{1} \, (\alpha_{22}^{1})^{*} \, \alpha_{23}^{1}$$

$$\alpha_{13}^{1} = \alpha_{13}^{0} + \alpha_{11}^{0} \, (\alpha_{11}^{0})^{*} \, \alpha_{13}^{0} = \emptyset + \Lambda(\Lambda)^{*} \, \emptyset = \emptyset$$

(también se ve directamente en el diagrama)

$$\alpha_{12}^{1} = \alpha_{12}^{0} + \alpha_{11}^{0} \, (\alpha_{11}^{0})^{*} \, \alpha_{12}^{0} = a + \Lambda(\Lambda)^{*} \, a = a$$

$$\alpha_{22}^{1} = \alpha_{22}^{0} + \alpha_{21}^{0} \, (\alpha_{11}^{0})^{*} \, \alpha_{12}^{0} = a + \emptyset \, (\Lambda)^{*} \, a = a$$

$$\alpha_{23}^{1} = \alpha_{23}^{0} + \alpha_{21}^{0} \, (\alpha_{11}^{0})^{*} \, \alpha_{13}^{0} = b + \emptyset \, (\Lambda)^{*} \, \emptyset = b$$

Luego:

$$\alpha_{13}^{2} = \emptyset + a(a^{*})b = aa^{*}b$$

Podríamos calcular α_{33}^{2} de manera análoga, pero ya

se ve directamente en el diagrama que saldrá $\alpha_{33}^{2}=b^{*}$

Por tanto,

$$\alpha_{13} = \alpha_{13}^{4} = \alpha_{13}^{3} = aa^{*}b + aa^{*}b(b^{*})^{*}b^{*} =$$

$$= aa^{*}b + aa^{*}bb^{*} = aa^{*}b(\Lambda + b^{*}) = \underline{aa^{*}bb^{*}}$$

2. Como sólo hay un estado final, q_2, la expresión regular será:

$$\alpha_{12} = \alpha_{12}^{4} = \alpha_{12}^{3} + \alpha_{14}^{3} \ (\alpha_{44}^{3})^{*} \ \alpha_{42}^{3}$$

Se ve sobre el diagrama que $\alpha_{42}^{3} = \emptyset$, por lo que ya no tenemos necesidad de calcularla, ni de calcular α_{14}^{3} ni α_{44}^{3}:

$$\alpha_{12} = \alpha_{12}^{3} = \alpha_{12}^{2} + \alpha_{13}^{2} \ (\alpha_{33}^{2})^{*} \ \alpha_{32}^{2}$$

Omitimos los cálculos de α_{12}^{2}, α_{33}^{2} y α_{32}^{2}, que se hacen por el mismo procedimiento de reducción, resultando:

$$\alpha_{12}^{2} = 1; \ \alpha_{13}^{2} = 10; \ \alpha_{33}^{2} = 10; \ \alpha_{32}^{2} = 1$$

(En los cálculos intermedios resulta una expresión de la forma $\emptyset^{*}$; no olvidar que $\emptyset^{*} = \Lambda$, no $\emptyset$).

Sustituyendo y aplicando las propiedades de las operaciones,

$$\alpha_{12} = 1 + 10(10)^*1 = (\Lambda+10(10)^*)1 = (10)^*1 = 1(01)^*$$

Dejamos como ejercicio la obtención de las expresiones regulares de los otros dos ejemplos. (En el ejemplo 3 hay dos estados finales, q_2 y q_4, por lo que la expresión será $\alpha_{12} + \alpha_{14} = \alpha^5_{12} + \alpha^5_{14} =$.....).

Como se habrá observado, el procedimiento es bastante engorroso para aplicarlo manualmente, pero es un algoritmo general que puede programarse para su ejecución automática. En la ejecución manual a veces puede simplificarse intuitivamente; así, cuando decíamos, en el ejemplo 1, que se ve directamente en el diagrama que $\alpha^3_{43} = \emptyset$, lo que nos evita muchos cálculos. De hecho, si el diagrama no es muy complicado, es preferible obtener la expresión regular por simple inspección.

4.2. Síntesis

4.2.1. Teorema de síntesis

Todo conjunto regular es un lenguaje aceptado por un reconocedor finito.

Hay publicadas varias demostraciones de este teorema, bastante laboriosas todas ellas, por lo que, a fin de no alargar excesivamente estos apuntes, nos permitimos no incluir ninguna, remitiendo al lector a las notas bibliográficas del Apartado 6. En cambio, nos parece interesante dar un algoritmo que permite deducir directamente el reconocedor de un conjunto regular denotado por su expresión regular. Para ello necesitamos introducir un nuevo concepto: el de derivadas de una expresión regular

4.2.2. Derivadas de una expresión regular

Consideremos una expresión regular, α, que designa a un conjunto regular L_R, $|\alpha| = L_R$, y consideremos el subconjunto de L_R formado por todas las cadenas de L_R que empiezan por un determinado símbolo, e. Definimos el cociente izquierdo de L_R por e, $L_R\backslash e$, como el conjunto resultante de suprimir e en todas esas cadenas:

$$L_R\backslash e = \{x \mid ex \in L_R\}$$

y definimos la derivada de α respecto al símbolo e, $D_e(\alpha)$, como la expresión regular de $L_R\backslash e$. (Es fácil ver que si L_R es regular, $L_R\backslash e$ también lo es).

Por ejemplo, sea $\alpha = abc + d + a^*c$, es decir, $L_R = |\alpha| = \{abc, d, c, ac, aac, \ldots\}$.

Entonces,

$$L_R\backslash a = \{bc, c, ac, \ldots\};\ L_R\backslash b = \emptyset,\ L_R\backslash c = \Lambda,\ L_R\backslash d = \Lambda,$$

y las derivadas serán las respectivas expresiones regulares:

$$D_a(\alpha) = bc + a^*c;\ D_b(\alpha) = \emptyset;\ D_c(\alpha) = \Lambda;\ D_d(\alpha) = \Lambda$$

Veamos algunas propiedades de las derivadas así definidas. De la definición es inmediato comprobar que:

a) $$D_{e_1}(e_2) = \begin{cases} \Lambda & \text{si } e_1 = e_2 \\ \emptyset & \text{si } e_1 \neq e_2 \end{cases}$$

b) $D_e(\Lambda) = D_e(\emptyset) = \emptyset$

c) $D_e(\alpha + \beta) = D_e(\alpha) + D_e(\beta)$

Ya no es tan inmediato calcular la derivada de la concatenación de dos expresiones regulares ni la del cierre de una expresión regular.

Supongamos que $\gamma=\alpha\beta$. Si $|\alpha|$ no contiene la cadena vacía, Λ, al suprimir "e" en las cadenas de $|\alpha\beta|$ resultarán las cadenas derivadas de α (es decir, todas aquellas que comiencen por "e", suprimiendo "e") concatenadas con las cadenas de β: $D_e(\alpha\beta) = [D_e(\alpha)]\beta$. Pero si $\Lambda \in (\alpha)$ entonces $|\alpha\beta|$ contiene también a todas las cadenas de β, por lo que habrá que añadir $D_e(\beta)$. Si introducimos la expresión $\delta(\alpha)$ tal que

$$\delta(\alpha) = \emptyset \text{ si } \Lambda \notin |\alpha|$$

$$\delta(\alpha) = \Lambda \text{ si } \Lambda \in |\alpha|$$

entonces podemos representar ambos casos en una sola expresión:

d) $D_e(\alpha\beta) = [D_e(\alpha)]\beta + \delta(\alpha)\, D_e(\beta)$

Finalmente, veamos cómo se calcula la derivada de la operación cierre. Sabemos que

$$|\alpha^*| = |\alpha|^* = \{\Lambda\} \cup |\alpha| \cup |\alpha||\alpha| \cup |\alpha||\alpha||\alpha|$$

por lo que

$$D_e(\alpha^*) = D_e(\Lambda) + D_e(\alpha) + D_e(\alpha\alpha) + D_e(\alpha\alpha\alpha)$$

$D_e(\Lambda) = \emptyset$.

Si $\Lambda \notin |\alpha|$

$$D_e(\alpha^*) = D_e(\alpha) + [D_e(\alpha)]\alpha + [D_e(\alpha)]\alpha\alpha + \ldots\ldots =$$

$$= D_e(\alpha)[\Lambda + \alpha + \alpha\alpha + \ldots.] = [D_e(\alpha)]\alpha^*$$

Si $\Lambda \in |\alpha|$ llegamos al mismo resultado. En efecto:

$$D_e(\alpha^*) = D_e(\alpha) + [D_e(\alpha)]\ \alpha + D_e(\alpha) + [D_e(\alpha)]\alpha\alpha + D_e(\alpha\alpha) + \ldots$$

que se reduce a la misma expresión anterior teniendo en cuenta la idempotencia de la suma. Luego:

e) $D_e(\alpha^*) = [D_e(\alpha)]\ \alpha^*$

Teniendo en cuenta estas cinco propiedades se puede calcular la derivada de cualquier expresión regular sin tener que formar previamente el conjunto cociente.

Hemos visto la derivación respecto de un símbolo, $e \in E$; la operación se puede extender a derivación respecto a una cadena definiendo:

$$D_\Lambda(\alpha) = \alpha$$

$$D_{xe}(\alpha) = D_e\ [D_x(\alpha)]$$

Se puede demostrar (por inducción sobre la longitud de las cadenas) que el conjunto de derivadas diferentes de una expresión regular, es decir, $\{D_x(\alpha) \mid x \in E^*\}$ es finito.

4.2.3. Algoritmo de síntesis.

Para construir un reconocedor en forma mínima del lenguaje denotado por la expresión regular α se puede seguir el siguiente procedimiento:

1. Calcular $\{D_x(\alpha) \mid x \in E^*\}$.
2. El estado inicial es $q_1 = \alpha$; los otros son las diferentes $D_x(\alpha)$.
3. La función de transición es $f(e,\alpha) = D_e(\alpha)$;

$$f(e,D_x(\alpha)) = D_{xe}(\alpha).$$

4. El conjunto de estados finales es $F = \{D_x(\alpha) \mid \Lambda \in \hat{D}_x(\alpha)\}$.

4.2.4. Ejemplos.

Para ilustrar la aplicación del algoritmo anterior vamos a tratar los mismos cuatro ejemplos de 1.2. Teníamos allí cuatro reconocedores; las expresiones regulares de los correspondientes lenguajes las obtuvimos en 3.3 y 4.1.3. Pues bien, ahora partiremos de esas expresiones y comprobaremos que, con el algoritmo de síntesis, llegamos a los diagramas originales.

Ejemplo 1.

$$\alpha = aa^*bb^* = (a)(a^*bb^*)$$

$$D_a(\alpha) = D_a(a)a^*bb^* + \delta(a)\ D_a(a^*bb^*) = \Lambda\ a^*bb^* + \emptyset\ D_a(a^*bb^*) = a^*bb^*$$

$$D_b(\alpha) = D_b(a)a^*bb^* + \delta(a)\ D_b(a^*bb^*) = \emptyset\ a^*bb^* + \emptyset\ D_b(a^*bb^*) = \emptyset$$

$$D_{aa}(\alpha) = D_a(a^*bb^*) = D_a(a^*)bb^* + \delta(a^*)\ D_a(bb^*) =$$

$$= D_a(a)a^*bb^* + \Lambda\emptyset = \Lambda a^*bb^* = a^*bb^* = D_a(\alpha)$$

Al repetirse la derivada, ya no seguimos derivando - respecto de a. Tampoco es necesario derivar $D_b(\alpha)$, puesto que, al ser $\emptyset$, cualquier derivada posterior será $\emptyset$. Nos queda pues por calcular $D_{ab}(\alpha)$.

$$D_{ab}(\alpha) = D_b(a^*bb^*) = D_b(a^*)bb^* + \delta(a^*)\ D_b(bb^*) =$$

$$= D_b(a)a^*bb^* + \Lambda[D_b(b)b^* + \delta(b)\ D_b(b^*)] =$$

$$= \emptyset\ a^*bb^* + \Lambda b^* + \emptyset\ \Lambda\ b^* = b^*$$

Calcularemos ahora las derivadas terceras a partir de $D_{ab}(\alpha)$ (puesto que ésta es la única derivada segunda que no es igual a ninguna anterior).

$$D_{aba}(\alpha) = D_a(b^*) = D_a(b)b^* = \emptyset\ b^* = \emptyset$$

$$D_{abb}(\alpha) = D_b(b^*) = D_b(b)b^* = b^* = D_{ab}(\alpha)$$

Al salir una $\emptyset$ y la otra repetida ya no es preciso - que sigamos derivando.

Según el algoritmo, el conjunto de estados será

$$Q = \{\alpha,\ D_a(\alpha),\ D_b(\alpha),\ D_{ab}(\alpha)\}$$

De las derivadas calculadas, la única que contiene la cadena vacía es $D_{ab}(\alpha) = b^*$, por lo que

$$F = \{D_{ab}(\alpha)\}$$

Y la función de transición será:

$$f(a,\alpha) = D_a(\alpha);\ f(b,\alpha) = D_b(\alpha)$$

$$f(a,D_a(\alpha)) = D_{aa}(\alpha) = D_a(\alpha)$$

$$f(b,D_a(\alpha)) = D_{ab}(\alpha)$$

$$f(a,D_b(\alpha)) = D_{ba}(\alpha) = D_b(\alpha)$$

$$f(b,D_b(\alpha)) = D_{bb}(\alpha) = D_b(\alpha)$$

$$f(a,D_{ab}(\alpha)) = D_{aba}(\alpha) = D_b(\alpha)$$

$$f(b,D_{ab}(\alpha)) = D_{abb}(\alpha) = D_{ab}(\alpha)$$

Podemos así dibujar el diagrama de la figura 4.5,que es el mismo de la figura 4.1, con $q_1 = \alpha$; $q_2 = D_a(\alpha)$; $q_3 = D_{ab}(\alpha)$; $q_4 = D_b(\alpha)$.

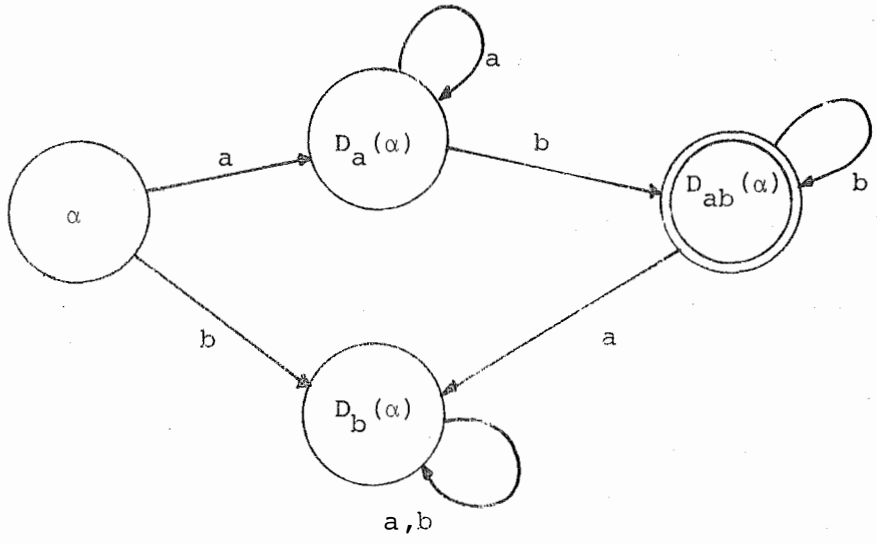


Fig. 4.5.

Ejemplo 2.

La expresión regular es $\alpha = 1(01)^*$. Calculemos las derivadas hasta que aparezcan repetidas:

$$D_0(\alpha) = D_0(1)(01)^* + \delta(1)\, D_0(01)^* = \emptyset$$

$$D_1(\alpha) = D_1(1)(01)^* + \delta(1)\, D_1(01)^* = (01)^*$$

$$D_{00}(\alpha) = D_{01}(\alpha) = \emptyset$$

$$D_{10}(\alpha) = D_0[(01)^*] = [D_0(01)]\,(01)^* = 1(10)^* = \alpha$$

$$D_{11}(\alpha) = D_1[(01)^*] = \emptyset$$

Por tanto,

$$Q = \{\alpha,\ D_0(\alpha),\ D_1(\alpha)\}$$

$$F = \{D_1(\alpha)\}$$

La función de transición será:

$$f(0,\alpha) = D_0(\alpha); \qquad f(1,\alpha) = D_1(\alpha)$$

$$f(0,D_0(\alpha)) = D_{00}(\alpha) = D_0(\alpha); \qquad f(1,D_0(\alpha)) = D_{01}(\alpha) = D_0(\alpha)$$

$$f(0,D_1(\alpha)) = D_{10}(\alpha) = \alpha; \qquad f(1,D_1(\alpha)) = D_{11}(\alpha) = D_0(\alpha)$$

Llegamos así a un diagrama de Moore como el de la figura 4.6 que, aparentemente, corresponde a un reconocedor diferente del original (Fig. 4.2.). No hay ningún error. Lo que ocurre es que este algoritmo que estamos aplicando nos da el reconocedor en forma mínima, y el de la figura 4.2 no lo está.

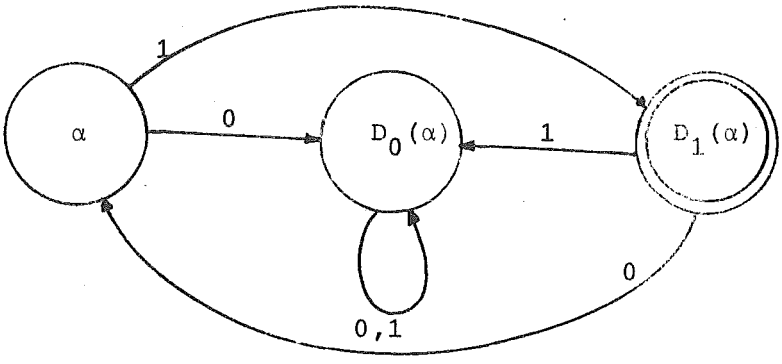


Fig. 4.6.

Invitamos al lector a aplicar el algoritmo de minimización explicado en el Capítulo 2 al AF de la figura 4.2 para ver que se llega al mismo AF de la figura 4.6.

Ejemplo 3.

$$\alpha = a + bc$$

Derivadas:

$$D_a(\alpha) = D_a(a) + D_a(bc) = \Lambda + \emptyset = \Lambda$$

$$D_b(\alpha) = D_b(a) + D_b(bc) = \emptyset + c = c$$

$$D_c(\alpha) = D_c(a) + D_c(bc) = \emptyset$$

$$D_{aa}(\alpha) = D_{ab}(\alpha) = D_{ac}(\alpha) = \emptyset = D_c(\alpha)$$

$$D_{ba}(\alpha) = D_a(c) = \emptyset = D_c(\alpha)$$

$$D_{bb}(\alpha) = D_b(c) = \emptyset = D_c(\alpha)$$

$$D_{bc}(\alpha) = D_c(c) = \Lambda = D_a(\alpha)$$

$$D_{ca}(\alpha) = D_{cb}(\alpha) = D_{cc}(\alpha) = \emptyset = D_c(\alpha)$$

$$Q = \{\alpha, D_a(\alpha), D_b(\alpha), D_c(\alpha)\}$$

$$F = \{D_a(\alpha)\}$$

Función de transición:

$f(a,\alpha) = D_a(\alpha)$; $f(b,\alpha) = D_b(\alpha)$; $f(c,\alpha) = D_c(\alpha)$

$f(a,D_a(\alpha))=D_{aa}(\alpha)=D_c(\alpha)$; $f(b,D_a(\alpha))=D_{ab}(\alpha)=D_c(\alpha)$; $f(c,D_a(\alpha))=$

$= D_{ac}(\alpha) = D_c(\alpha)$

$f(a,D_b(\alpha))=D_{ba}(\alpha)=D_c(\alpha)$; $f(b,D_b(\alpha))=D_{bb}(\alpha)=D_c(\alpha)$; $f(c,D_b(\alpha))=$

$= D_{bc}(\alpha) = D_a(\alpha)$

$f(a,D_c(\alpha))=D_{ca}(\alpha) =D_c(\alpha)$; $f(b,D_c(\alpha))=D_{cb}(\alpha)=D_c(\alpha)$; $f(c,D_c(\alpha))=$

$= D_{cc}(\alpha) = D_c(\alpha)$

Con esto, resulta el diagrama de la figura 4.7, que es distinto del original de la figura 4.3. La explicación es la misma del ejemplo anterior: el de la figura 4.3 no está en forma mínima (q_2 y q_4 son equivalentes)

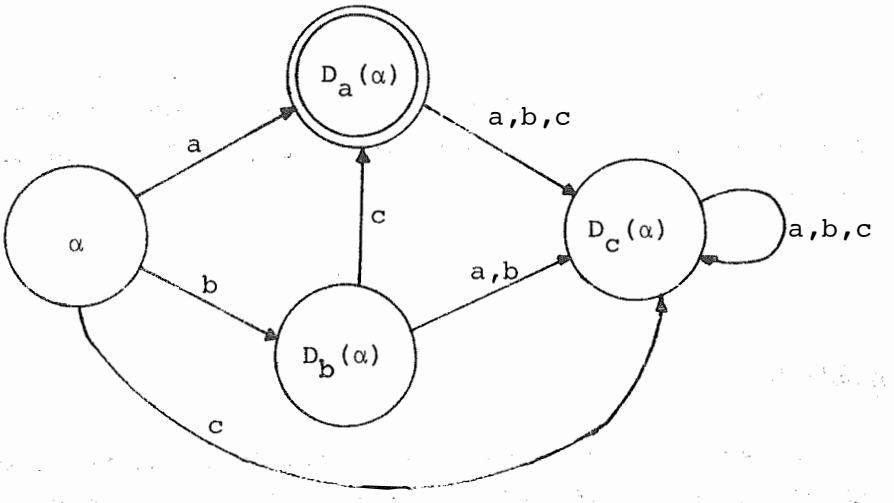


Fig. 4.7.

Ejemplo 4.

$$\alpha = (1 + 2)^* 3$$

Derivadas:

$$D_1(\alpha) = D_1\left[(1+2)^*\right] 3 + \delta\left[(1+2)^*\right] D_1(3) =$$

$$= \left[D_1(1+2)\right](1+2)^* 3 + \Lambda\, \emptyset = (1+2)^* 3 = \alpha$$

$$D_2(\alpha) = \ldots = (1+2)^* 3 = \alpha$$

$$D_3(\alpha) = D_3\left[(1+2)^*\right] 3 + \delta\left[(1+2)^*\right] D_3(3) = \emptyset + \Lambda\Lambda = \Lambda$$

$$D_{31}(\alpha) = D_{32}(\alpha) = D_{33}(\alpha) = \emptyset$$

$$Q = \{\alpha,\ D_3(\alpha),\ D_{31}(\alpha)\}$$

$$F = \{D_3(\alpha)\}$$

Calculando los valores de la función de transición se llega a un diagrama idéntico al de la figura 4.4, con $q_1 = \alpha$, $q_2 = D_3(\alpha)$, $q_3 = D_{31}(\alpha)$.

5. RESUMEN.

Orientándonos ya hacia las relaciones entre lenguajes y autómatas, hemos definido un reconocedor finito como un tipo particular de autómata finito en el que existe un estado inicial fijo y el alfabeto de salida consta sólo de dos símbolos, correspondientes a la aceptación o no aceptación de la cadena de entrada.

Hemos demostrado la condición general que debe cumplir un lenguaje para que todas sus cadenas sean aceptadas por un reconocedor finito: que la relación de congruencia derecha inducida por el lenguaje tenga índice finito (ese índice es, precisamente, igual al número de estados del reconocedor).

Se han definido los conjuntos regulares, que, según los teoremas de análisis y síntesis, son justamente los lenguajes que pueden ser aceptados por un reconocedor finito. Las expresiones regulares constituyen un metalenguaje para describir de una manera cómoda a los conjuntos regulares. El algoritmo de análisis permite, dado un reconocedor finito, deducir la expresión regular del lenguaje que acepta ese reconocedor. A la inversa, hemos visto un algoritmo de síntesis mediante el cual se llega al diagrama de Moore de un reconocedor finito minimizado correspondiente a una expresión regular dada.

6. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

Los conceptos de conjunto regular y expresión regular, así como los teoremas de análisis y síntesis, se deben a KLEENE (1.956).

La demostración del teorema de análisis que hemos seguido es la de McNAUGHTON y YAMADA (1.960).

Para el teorema de síntesis existen varias demostraciones. Quizá la más utilizada es la debida a RABIN y SCOTT (1.959), que precisa de la introducción de un tipo especial de autómata que no tiene interpretación física: el autómata no determinista (o "posibilístico", como prefiere llamarlo ARBIB (1.969), ya que no interviene la idea de probabilidad). Una generalización de este autómata es el llamado sistema de transición (OTT y FEINSTEIN, 1.961), que permite llegar a un algoritmo bastante cómodo para la síntesis. Sin embargo nos ha parecido que el algoritmo de síntesis más manejable es el basado en el trabajo de BRZOZOWSKI (1.962, 1.965), que introdujo el concepto de derivada de una expresión regular.

Las anteriores referencias tienen un interés más bien histórico. Para ampliar este Capítulo es preferible dirigirse a libros de carácter general, como los ya citados en el Capítulo 2, o el de HOPCROFT y ULLMAN (1.969).

Los cuatro ejemplos que hemos utilizado reiteradamente están adaptados de SCALA y MINGUET (1.974).

7. EJERCICIOS

1. Dada la tabla de transición

	a	b	c
q_1	q_2	q_6	q_1
q_2	q_3	q_5	q_1
q_3	q_3	q_4	q_1
q_4	q_7	q_7	q_1
q_5	q_6	q_5	q_1
q_6	q_7	q_6	q_1
(q_7)	q_7	q_7	q_1

correspondiente a un reconocedor finito, hallar la expresión regular correspondiente al lenguaje aceptado por ese reconocedor.

2. Aplicar el algoritmo de síntesis a la expresión regular encontrada en el ejercicio anterior.

3. Aplicar el algoritmo de análisis al reconocedor de la cadena 010 (Cap.2, Apartado 4.5.2) y al reconocedor de 321 (Cap. 2, Ejercicio 10). En esos ejemplos se utilizó "reconocedor" con un sentido distinto del definido en este Capítulo. ¿Cuáles son las diferencias?.

4. Dar una expresión regular correspondiente al detector de paridad par y otra al detector de paridad impar, y aplicar el algoritmo de síntesis para obtener los correspondientes reconocedores.

5. Diseñar un circuito secuencial que dé una salida "1" cuando la cadena de entrada tenga un número de "unos" congruente a 0 (mod 2). (La expresión regular es $\alpha = 0^*(10^*10^*)^*$)

6. Considérese un reconocedor finito definido por:

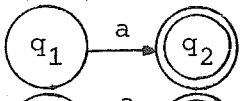
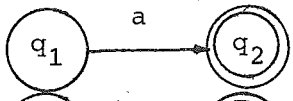
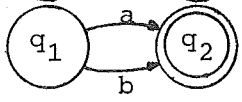
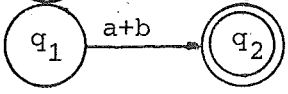
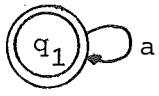
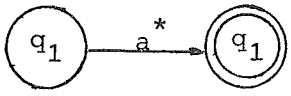
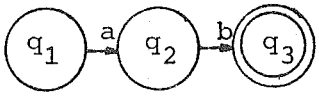
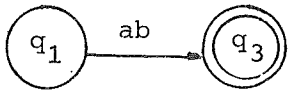
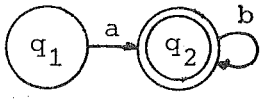
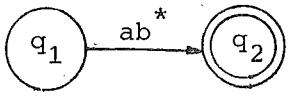
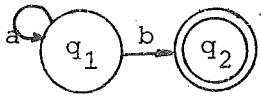
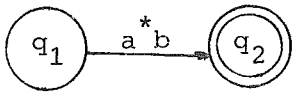
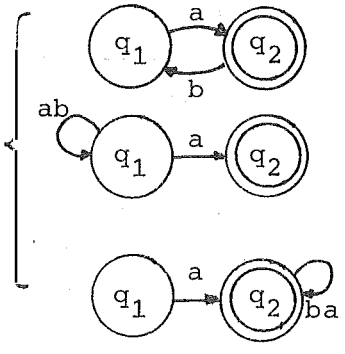
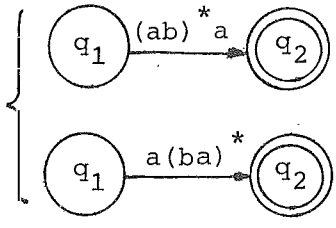
$$E = \{a,b,c,d\}; \quad Q = \{q_1,q_2,q_3,q_4\}; \quad F = \{q_4\};$$

q \ e	a	b	c	d
q_1	q_2	q_3	q_4	q_4
q_2	q_3	q_2	q_4	q_4
q_3	q_3	q_3	q_3	q_3
(q_4)	q_3	q_3	q_4	q_4

Hallar una expresión regular de las cadenas aceptadas por el reconocedor, y diseñar un circuito secuencial para la realización del reconocedor con biestables JK.

Los dos ejercicios siguientes, tomados de BOOTH (1.967), desarrollan una técnica gráfica para obtener la expresión regular directamente del diagrama de Moore.

7. Las siguientes expresiones regulares básicas describen a los conjuntos regulares de los reconocedores correspondientes. (El estado inicial es siempre q_1):

	Expresión regular	Diagrama	Diagrama reducido
1)	a		
2)	a + b		
3)	a^*		
4)	ab		
5)	ab^*		
6)	a^*b		
7)	$(ab)^*a=a(ba)^*$		

a) Comprobar que las expresiones regulares corresponden efectivamente a los diagramas.

b) Utilizar esas relaciones para obtener la expresión regular del reconocedor dado por la tabla:

	a	b
q_1	q_1	q_2
q_2	q_4	q_5
q_3	q_5	q_1
q_4	q_3	q_2
(q_5)	q_4	q_5

8. Utilizando como punto de partida las relaciones básicas establecidas en el ejercicio anterior, desarrollar una técnica general gráfica de análisis para deducir la expresión regular de cualquier reconocedor finito.

CAPITULO 5.

AUTOMATAS ESTOCASTICOS

1. INTRODUCCIÓN

Los AF estudiados hasta ahora son modelos de sistemas discretos con memoria (sistemas secuenciales). Implícitamente, venimos suponiendo que tales sistemas son deterministas, es - decir, dado un estado y un símbolo de entrada, el estado si-- guiente y el símbolo de salida vienen determinados por las - - funciones f y g respectivamente. Pero se dan situaciones cuya complejidad funcional y la imposibilidad de analizar todos - los factores que intervienen hacen que sea preferible modelar las a partir de la idea de probabilidad. Tal ocurre, por ejemplo, cuando la fiabilidad de los componentes es pequeña, o - cuando se quiere modelar órganos o procesos de los sistemas - vivos. Así, si observamos que, estando en el estado q_1 y recibiendo entrada e_1 el sistema pasa al estado q_2 un 80% de las veces y al q_3 un 20%, diremos que $f(e_1,q_1) = q_2$ con probabilidad 0,8 y $f(e_1,q_1) = q_3$ con probabilidad 0,3. Esta es la idea básica del autómata probabilista o estocástico.

Pero quizá el mayor interés de los autómatas estocásticos está en su capacidad adaptativa y de aprendizaje. No podemos dejar de exponer en este Tema, aunque sea en líneas muy generales, la teoría básica de los autómatas de aprendizaje, cuyas aplicaciones son ya numerosas, pero se prevé que lo - - sean mucho más: control adaptativo, reconocimiento de formas, reconocimiento de lenguajes naturales, etc.

Finalmente, también damos unas ideas sobre un concep-

to algo antiguo que no ha perdido actualidad: las redes de - neuronas formales.

2. Autómatas estocásticos

2.1. Definición

Un autómata estocástico o probabilista es una quíntupla

$$A_p = \langle E, S, Q, P, h \rangle,$$

donde:

E, S, Q son, como en los autómatas deterministas, los alfabetos de entrada y salida y el conjunto de estados, que - supondremos finito:

$$Q = \{q_1, q_2, \ldots\ldots, q_n\}$$

h: $Q \to S$ es la función de salida (consideraremos sólo autómatas de tipo Moore), que se supone determinista.

P: $E \times S \to [0,1]^n$ es la función de probabilidades de transición:

$$P(e,q) = (p_1(e,q),\ p_2(e,q), \ldots, p_n(e,q)),$$

donde $0 \leq p_i(e,q) \leq 1$ es la probabilidad de que, estando en - el estado q y recibiendo la entrada e, se pase al estado q_i. (Deberá cumplirse que

$$\sum_{i=1}^{n} p_i(e,q) = 1).$$

Si consideramos todos los estados en que puede encontrarse el autómata cuando recibe una entrada e, podemos definir una matriz de probabilidades de transición, M(e):

$$M(e) = \begin{bmatrix} p_1(e,q_1) \ldots\ldots p_n(e,q_n) \\ \vdots \qquad\qquad \vdots \\ p_n(e,q_1) \ldots\ldots p_n(e,q_n) \end{bmatrix}$$

de modo que el elemento $m_{ij} = p_j(e,q_i)$ de M(e) es la probabilidad de pasar del estado q_i al q_j bajo la acción de la entrada e.

Si $x = e_1 e_2 \ldots e_m \in E^*$, es fácil demostrar, por inducción sobre m, que $M(x) = (p_j(x,q_i)) = M(e_1).M(e_2) \ldots\ldots M(e_m)$.

Un AF determinista es un caso particular del estocástico. En el caso estocástico, $m_{ij} \in [0,1]$, y en el determinista $m_{ij} \in \{0,1\}$. En cualquier caso, la suma de los elementos de cada columna en todas las matrices M debe ser la unidad.

2.2. Ejemplo.

Sea un autómata estocástico definido por

$E = S = \{0,1\}$

$Q = \{q_1, q_2\}$

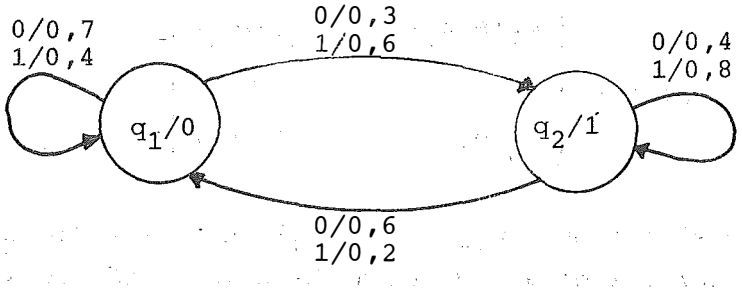
$h(q_1) = 0; \quad h(q_2) = 1$

$P(0,q_1) = (0,7;\ 0,3); \quad P(1,q_1) = (0,4;\ 0,6)$

$P(0,q_2) = (0,6;\ 0,4); \quad P(1,q_2) = (0,2;\ 0,8)$

Podemos representar este autómata por un diagrama de

Moore (Fig. 5.1), indicando sobre cada transición la probabilidad asociada.



Las matrices de probabilidades de transición correspondientes a los dos símbolos de entrada son:

$$M(0) = \begin{bmatrix} 0,7 & 0,3 \\ 0,6 & 0,4 \end{bmatrix} ; \qquad M(1) = \begin{bmatrix} 0,4 & 0,6 \\ 0,2 & 0,8 \end{bmatrix}$$

Si por ejemplo estamos interesados en la respuesta a la cadena x = 011, tendremos:

$$M(011) = \begin{bmatrix} 0,7 & 0,3 \\ 0,6 & 0,4 \end{bmatrix} \cdot \begin{bmatrix} 0,4 & 0,6 \\ 0,2 & 0,8 \end{bmatrix}^2 = \begin{bmatrix} 0,268 & 0,732 \\ 0,264 & 0,736 \end{bmatrix}$$

Así, la probabilidad de pasar de q_1 a q_2 bajo la acción de x = 011 es 0,732. Al mismo resultado se llega si se consideran las cuatro posibilidades existentes para pasar de q_1 a q_2 con una cadena de longitud 3:

$\xrightarrow{0} q_1 \xrightarrow{1} q_1 \xrightarrow{1} q_2$: probabilidad = 0,7 x 0,4 x 0,6 = 0,168

$\xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{1} q_2$: probabilidad = 0,7 x 0,6 x 0,8 = 0,336

$\xrightarrow{0} q_2 \xrightarrow{1} q_2 \xrightarrow{1} q_2$: probabilidad = 0,3 x 0,8 x 0,8 = 0,192

$\xrightarrow{0} q_2 \xrightarrow{1} q_1 \xrightarrow{1} q_2$: probabilidad = 0,3 x 0,2 x 0,6 = 0,036

Probabilidad total = 0,732

2.3. Reconocedores estocásticos

Siguiendo la misma línea del Capítulo 4, podemos definir un reconocedor estocástico como

$$R_p = \langle E, Q, q_1, P, F \rangle ,$$

donde q_1 es el estado designado como inicial y $F \subset Q$ es el conjunto de estados finales. (Algunos autores introducen tambien la indeterminación de tomar un estado inicial u otro, y, en lugar de q_1, incluyen un conjunto, $\Psi = \{\psi_1, \psi_2, \ldots \psi_n\}$, que representa las probabilidades asociadas a cada estado inicialmente).

El estudio de los lenguajes aceptados por reconocedores estocásticos es más complicado que en el caso determinista, ya que una misma cadena puede ser aceptada por el reconocedor en unas ocasiones y rechazada en otras. La probabilidad de que $x \in E^*$ sea aceptada será

$$p(x) = \sum_i p_i(q_1, x)$$

Así, en el ejemplo anterior, si q_1 es el estado inicial y q_2 el final, p(011) = 0,732.

Se define el lenguaje aceptado por un reconocedor estocástico haciéndolo depender de un parámetro, λ , llamado punto de corte del lenguaje:

$$L_{RP}(\lambda) = \{x \in E^* | q(x) > \lambda\}$$

y se demuestra que tales lenguajes son una generalización de los conjuntos regulares.

3. AUTÓMATAS ESTOCÁSTICOS DE ESTRUCTURA VARIABLE.

Un autómata estocástico o probabilista de estructura variable, APEV, es una séxtupla

$$APEV = \langle E, S, Q, P, h, A \rangle,$$

donde E, S, Q, P, h tienen el mismo significado anterior, y A es un algoritmo llamado esquema de actualización o esquema de refuerzo que genera P_{t+1} a partir de P_t, s_t y e_t. Así en un APEV las probabilidades de transición no son siempre las mismas, sino que van evolucionando en función de su valor anterior, de la respuesta última y de la entrada. *Esta característica es la que confiere al APEV la capacidad de adaptarse y de aprender.*

4. AUTÓMATAS DE APRENDIZAJE

4.1. Concepto de aprendizaje.

Las palabras "adaptación" y "aprendizaje" tienen diferente significado según quién las utiliza. En psicología representan conceptos bastante diferentes, mientras que en ingeniería suelen tomarse como sinónimos.

Aquí nos limitaremos a definir el aprendizaje como la *capacidad de un sistema para mejorar la probabilidad de emitir una respuesta correcta como resultado de la interacción con su entorno*. Esto implica que el entorno tiene capacidad de evaluación de las respuestas del sistema. Un esquema general de un proceso de aprendizaje es el de la figura 5.2.

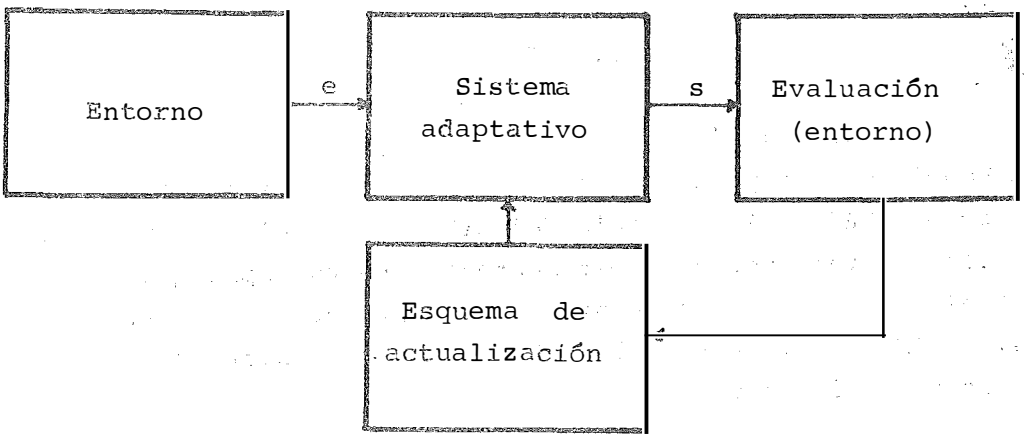


Fig. 5.2.

4.2. Aprendizaje en un APEV

De acuerdo con lo dicho, un APEV en interacción con un entorno (Fig. 5.3) puede presentar un comportamiento adap-

tativo que le confiera capacidad de aprendizaje. *Así, un autómata de aprendizaje será un APEV que opera en un entorno y actualiza sus probabilidades de transición de acuerdo con las entradas recibidas del entorno para mejorar su comportamiento en algún sentido especificado.*

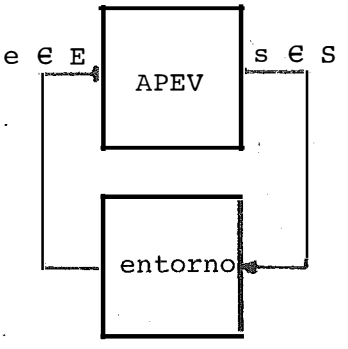


Fig. 5.3.

En psicología, un autómata de aprendizaje puede ser un modelo del comportamiento de un organismo bajo estudio, donde el APEV será el modelo del organismo y el entorno estará representado por el experimentador. En una aplicación de ingeniería, tal como el control de un proceso, el controlador es el APEV, y el resto del sistema, con sus incertidumbres, constituye el entorno.

Las respuestas del entorno son frecuentemente binarias, es decir, E = {0,1}, y a una de ellas se le llama "respuesta de premio" y a la otra "respuesta de castigo".

La utilización de un autómata de aprendizaje sólo tiene interés cuando el comportamiento del entorno es desconocido, puesto que si no fuera así, un AF determinista nos resuelve el problema. Generalmente se supone que el entorno es aleatorio, y la probabilidad de que produzca salida "1" dependerá

de su entrada, es decir, de la respuesta s del autómata. Así, si S tiene r elementos, tendremos r probabilidades C_i (i = 1,2,...,r) de que el entorno de salida "1". A estas probabilidades se les llama probabilidades de castigo.

Para juzgar el grado de aprendizaje se define un castigo medio recibido por el autómata: en un instante t, si el autómata produce la salida s_i con probabilidad $p_i(t)$, el castigo medio condicionado a P(t) es:

$$M(t) = E\left[e(t)\,|\,P(t)\right] = \sum_{i=1}^{r} p_i(t)C_i$$

Si suponemos que en t = 0 el autómata escogerá una salida u otra con igual probabilidad se tendrá

$$M(0) = \frac{\Sigma C_i}{r}$$

El uso del término "aprendizaje" sólo tiene sentido - si M(t) se va haciendo menor que M(0). Según cómo sea esa tendencia se definen unos tipos u otros de autómatas, y se estudian los esquemas de actualización necesarios para conseguirlos.

5. NEURONAS FORMALES

Se ha visto cómo todo AF puede realizarse físicamente mediante elementos electrónicos correspondientes a funciones lógicas combinacionales y a funciones de memoria. Otra posible realización es mediante "neuronas formales". Una neurona

formal es un modelo, más o menos simplificado (generalmente muy simplificado), de la célula nerviosa de los organismos superiores. El más sencillo de tales modelos es la neurona formal de McCulloch - Pitts, que se puede definir como un AF en el que

a) $S = Q = \{0,1\}$. ("0" se llama "estado inhibido" y "1", "estado excitado").

b) $E = \{0,1\}^n$, es decir, hay n hilos de entrada binarios; cada uno de ellos, e_i, lleva asociado un peso, ω_i. Si $\omega_i > 0$ la entrada se dice que es "excitadora", y si $\omega_i < 0$, "inhibidora".

c) Las funciones f y h son:

$$(1)\quad q(t+1) = \begin{cases} 1 \text{ si } \sum_{i=1}^{n} \omega_i e_i \geq K \\ 0 \text{ si } \sum_{i=1}^{n} \omega_i e_i < K, \end{cases}$$

siendo K una constante de la neurona, llamada "umbral"

$$(2)\quad s(t) = q(t)$$

La representación gráfica puede ser la de la figura 5.4.

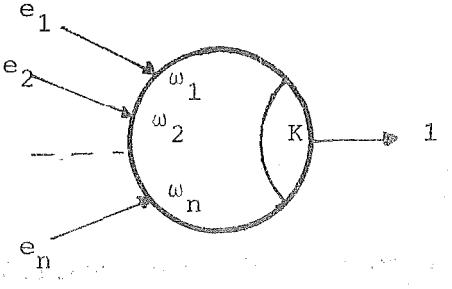


Fig. 5.4.

Obsérvese que la neurona formal así definida corresponde exactamente a la puerta de umbral que se definió en el Tema "Lógica" (Cap.6, Apartado 5). La única diferencia está en que ahora suponemos que existe un retardo en el interior de la neurona. De hecho, la lógica de umbral se ha desarrollado para formalizar el estudio de las redes neuronales.

Una red neuronal es una combinación de neuronas formales interconectadas. Ya McCulloch y Pitts demostraron que todo AF admite una realización física mediante una red neuronal.

Neuronas formales más complicadas se obtienen considerando los pesos ω_i y/o el umbral K aleatorios. Esto conduce a redes neuronales aleatorias, muy interesantes desde el punto de vista de modelación de sistemas neurofisiológicos, pero que también encuentran aplicaciones en ingeniería. Así, una de las primeras máquinas para reconocimiento y clasificación de formas, el perceptrón, era una red de ese tipo.

6. NOTAS HISTÓRICA Y BIBLIOGRÁFICA

El concepto de autómata estocástico fue introducido al comienzo de los años 60 en USA (RABIN, 1.963) pensando más bien en el objetivo de mejorar la fiabilidad de los circuitos secuenciales (WINOGRAD y COWAN, 1.963) que en su aplicación a sistemas de aprendizaje.

El interés por los sistemas de aprendizaje tiene su origen en la URSS. TSETLIN (1.961) introduce la idea de utilizar AF deterministas en un entorno aleatorio como modelos de aprendizaje. Algo más tarde, VARSHAVSKII y VORONTSOVA (1.963) demuestran que el número de estados se reduce si se utilizan

autómatas estocásticos con actualización de las probabilidades de transición (es decir, con estructura variable).

Una buena referencia para adquirir una visión global del estado de los estudios sobre autómatas de aprendizaje es el artículo de NARENDRA y THATHACHAR (1.974).

El primer modelo de neurona fue el propuesto por McCULLOCH y PITTS (1.943). Posteriormente, VON NEUMANN(1.956) introdujo la noción de probabilidad en la neurona formal para demostrar que, gracias a la redundancia, se pueden sintetizar sistemas muy fiables a partir de elementos poco fiables. Actualmente se estudian redes neuronales a base de neuronas formales más complicadas y realistas que las McCulloch y Pitts (umbral y pesos aleatorios, introducción de un período refractario, etc.).

Las redes neuronales se suelen utilizar como herramienta para el estudio matemático de los autómatas (ARBIB, 1.969) y tambien como modelos de interés neurofisiológico (CALLE, 1.977). Dos libros de divulgación recomendables sobre estos puntos son los de ARBIB (1.965) y SINGH (1.976).

REFERENCIAS BIBLIOGRAFICAS

ARBIB, M.A. Theories of abstract automata. Prentice-Hall, Englewood Cliffs, N.J., 1.969.

ARBIB, M.A. Brains, Machines and Mathematics. McGraw-Hill, New York, 1.965 (Versión en castellano: Cerebros, máquinas y matemáticas. Alianza Ed., Madrid, 1.976).

ASHBY, R. An introduction to Cybernetics. Chapman et Hall, New York, 1.956. (Versión en castellano: Introducción a la cibernética, Nueva Visión, Buenos Aires, 1.976, 3ª. ed.).

BARTEE, T.C., LEBOW, I.L. y REED, I.S. Theory and design of digital machines. Mac Graw-Hill, New York, 1.962.

BOOTH, T.L. Sequential machines and automata theory. John Wiley, New York, 1.967.

BRZOZOWSKI, J.A. A survey of regular expressions and their applications. IRE Trans. Elec. Comp., EC-11,324 - 325 (1.962).

BRZOZOWSKI, J.A. Regular expressions for linear sequential circuits. IEEE Trans. Elec. Comp., EC-14, 148-156 (1.965).

CALLE, J.A. Sistema nervioso y sistemas de información. Ed. Pirámide, Madrid,1.977.

GILBERT, W.J. Modern algebra with applications. John Wiley, New York, 1.976.

HARRISON, M.A. Introduction to switching and automata theory. Mc Graw-Hill, New York, 1.965.

HOPCROFT, J.E. y ULLMAN, J.D. Formal languages and their relation to automata. Addison - Wesley, Reading, Mass., 1.969.

HUFFMAN, D.A. The synthesis of sequential switching circuits. J. Franklin Inst., vol. 257, números 3,4 (1.954), pp. 161-190, 275-303.

KLEENE, S.C. Representation de events in nerve nets and finite automata. En Automata Studies, Princeton Univ. - Press, Princeton, N.J., 1.956.

KUBUSCH,A. y CARDENAS, E. Informática. En Fronteras de la Ciencia, U.N.E.D., Madrid, 1.977.

MANDADO, E. Sistemas electrónicos digitales. Marcombo - Boixareu, Barcelona, 1.977 (3^a. ed.).

McCULLOCH, W. y PITTS, W. A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophysics, 5 (1.943), 115-133

McNAUGHTON, R. y YAMADA, H. Regular expressions and state - - graphs for automata. IRE Trans. Elec. Comp., EC-9 (1.960), 39-47.

MEALY, G.H. A. method for synthesizing sequential circuits. - Bell System Tech., vol 34 (1.955), pp.1.045-1.079

MOORE, E.F. Gendanken-Experiments on sequential machines. Automata Studies, Annals of Mathematical Studies,No. - 34. Princeton Univ. Press, Princeton, N.J., 1.956, pp. 129-153.

MUÑOZ, E. Circuitos electrónicos digitales II. Dpto. Publi-

caciones E.T.S.I.T., Madrid, 1.977.

NAGLE, H.T., CARROL, B.D. y IRWIN, J.D. An introduction to computer logic. Prentice-Hall, Englewood Cliffs, N.J. 1.975.

NARENDRA, K.S. y THATHACHAR, M.A.L. Learning automata - A survey. IEEE Trans. Systems, Man and Cybernetics, SMC-4 (1.974), 323-334

OTT, G. y FEINSTEIN, N. Design of sequential machines from their regular expressions. Journal Assoc. Comp. Mach., vol 8, n° 4 (1.961), 585-600.

RABIN, M.O. Probabilistic automata. Information and Control, 6 (1.963), n° 3, 230-245.

RABIN, M.O. y SCOTT, D. Finite automata and their decision problems. IBM Journal Res. Dev., vol.3, n° 2 (1.959), 114-125.

SCALA, J.J. y MINGUET, J.M. Informática II. Unidad didáctica 2. Universidad Nacional de Educación a Distancia, Madrid, 1.974.

SINGH, J. Teoría de la información, del lenguaje y de la cibernética. Alianza Ed., Madrid, 1.976.

TSETLIN, M.L. Sobre el comportamiento de autómatas finitos en entornos aleatorios. Automática y Telemecánica, 22 (1.961), 1.345-1.354.

VARSAAVSKII, V.I. y VORONTSOVA, I.P. Sobre el comportamiento de autómatas estocásticos con estructura variable. Automática y Telemecánica, 24 (1.963), 353-360.

VON NEUMANN, J. Probabilistic logic and the synthesis of reliable organisms from unreliable components. En Automata Studies, Princeton Univ. Press, Princeton, N. J., 1.956.

WINOGRAD, S y COWAN, J.D. Reliable computation in the presence of noise, M.I.T. Press, Cambridge, Mass., 1.963.

TEMA 3

ALGORITMOS, PROGRAMACION ESTRUCTURADA Y MAQUINAS DE TURING

FERNANDO SAEZ VACAS

INDICE

Pág.

Pág.

Pág.

CAPITULO 1

ALGORITMOS

0. INTRODUCCIÓN.

La palabra *algoritmo* procede del apellido latinizado - de un matemático árabe, Mohamed ibn Mûsâ al - Khowârizmî (o al - Khârezmi) que, en 820 y 825 d. de C., escribió respectivamente dos tratados, el primero de cálculo con los números hindúes y el segundo de resolución de ecuaciones. La deformación del - título de esta última obra ha originado el nombre de álgebra - con el que se conoce a la rama de las matemáticas consagrada - al cálculo literal.

En nuestro siglo, no solamente se ha hecho más frecuente el uso del término *algoritmo*, sino que su contenido, su significación precisa, ha sido explorado con ahinco. En la técnica de los computadores es palabra de empleo cotidiano, si bien es necesario advertir que no demasiados profesionales de los - computadores la conocen y emplean en su sentido profundo. Lo - cierto es que la aparición de los computadores con sus extraordinarias posibilidades de cálculo y de almacenamiento de información ha impulsado increiblemente el estudio de los algoritmos. Tanto es así que a veces es difícil percibir que pueden - distinguirse *dos campos de interés*: uno de índole fundamental en matemáticas, en donde se plantea la pregunta de *si tienen - solución ciertos tipos de problemas, lo que es equivalente a - preguntarse si existe un algoritmo que conduzca siempre a una solución para esa clase de problema*. El otro campo tiene relación con la misma existencia de las máquinas computadoras (en un sentido muy general, de los autómatas) y en él se investi--

gan *los problemas que surgen de la aplicación de tales máquinas al procesamiento de algoritmos*. Ambos campos están muy interrelacionados, hasta el punto de que no son fáciles de delimitar sus fronteras. Por ejemplo, piénsese que la computabilidad o *calculabilidad* (aludiendo al primer campo) de determinados problemas puede investigarse gracias precisamente a la potencia de cálculo aritmético o lógico de los computadores digitales.

Nuestro enfoque de este tema hará más hincapié en el segundo campo de interés. Trataremos de dar una visión lo más completa que sea posible -dentro de un criterio de simplificación-, del concepto de algoritmo, de su relación con la programación de los computadores y, en especial, de su planteamiento en términos de un autómata muy particular, la máquina de Turing.

El presente capítulo se dedica básicamente a definir el concepto de algoritmo. Primero empieza con unas definiciones diversas no totalmente coincidentes que se contrastarán. Se pasará a continuación a una definición formal en teoría de conjuntos que comprende y precisa todas las anteriores.

De la definición se extraen unas propiedades o condiciones que debe cumplir todo algoritmo, a las que se añaden propiedades que se puede desear que cumplan los "buenos algoritmos"

El capítulo prepara ya el terreno para hablar de programas y de máquinas destacando en la definición formal aquellos elementos, como el número de orden de una ejecución y la existencia misma de los estados, que prefiguran la existencia de determinadas condiciones generales en, por ejemplo, cualquier máquina ejecutora de algoritmos.

1. DEFINICIONES DE ALGORITMO.

No hay una, sino muchas definiciones de algoritmo. - - Aquí recuadramos varias de distintos autores, todas (¡y no es casualidad!) tomadas de libros sobre computación o computado-- res.

Definiciones de algoritmo

1. Lista de instrucciones que especifican una secuen-- cia de operaciones que darán la contestación a cual quier problema de un tipo determinado.

(Trakhtenbrot, 1.960)

2. Conjunto de reglas que define de manera precisa una secuencia de operaciones tales que cada regla es - efectiva y definida y tal que la secuencia termina en un tiempo finito.

(Stone, 1.972)

3. Sucesión finita de prescripciones potencialmente - ejecutables expresadas en un lenguaje definido que estipula como ejecutar un cierto encadenamiento de operaciones para resolver todos los problemas de un cierto tipo dado.

(Corge, 1.975)

4. Sistema de reglas que permiten obtener una salida - específica a partir de una entrada específica. Cada paso debe estar definido exactamente, de forma que pueda traducirse a lenguaje de computador.

(Knuth, 1.977)

Lo primero que sorprende es que las definiciones difieren no poco entre sí, aunque tal vez sea esta una cuestión de apariencias. Esto deja ya suponer que no son muy precisas. Pero fijémonos en las semejanzas y no en las diferencias.

1°. Hay unas *reglas*, o instrucciones, o prescripciones.

2°. Tales reglas, instrucciones, etc, especifican una *secuencia* (encadenamiento) *de operaciones* o pasos.

3°. Aunque de forma muy implícita (excepto en la definición de Knuth) las operaciones se supone han de ser llevadas a cabo por un *agente ejecutor*, máquina o ser vivo, por sí o a través de otros agentes. Agente que es el destinatario de las instrucciones.

4°. Más implícitamente aún, pero contenido en las definiciones, se establece que la secuencia de operaciones tiene una *duración* , que podrá ser tan larga como se quiera, pero ha de ser *finita*.

Con estos elementos, que constituyen un común denominador de lo que hemos podido descubrir hasta ahora sobre la conceptualización moderna de los algoritmos, vemos que es posible elaborar algoritmos para resolver muchas clases de problemas. Así, por ejemplo, encontrar el máximo común divisor de dos números enteros, investigar si una palabra determinada figura en una tabla de palabras almacenada en la memoria de un computador, jugar una partida de ajedrez o tornear una pieza complicada.

La amplitud del campo de los problemas es tan grandiosa que cualquiera percibe la dificultad de aprehender la noción de algoritmo si uno se sitúa en medio de la diversidad de los problemas y la diversidad de los agentes ejecutores. Es obligatorio reducir el ámbito de atención e investigar el asunto como un proceso intelectual independiente del problema espe

cífico, por una parte. De ahí se desprende que, si bien hay algoritmos numéricos y no numéricos, en última instancia todos pueden reducirse, a la especificación de operaciones sobre símbolos. Y por otra, es obligatorio independizarse en lo posible del agente ejecutor de estas operaciones simbólicas y para ello, una solución ha consistido en definir un agente ejecutor único (veremos que será la máquina de Turing), al cual podrían reducirse en última instancia todos los demás.

2. ALGORITMOS Y MÁQUINAS.

La resolución de un problema implica un proceso de varias etapas que, a grandes rasgos, son las que expresa el diagrama de flujo de la figura 1.

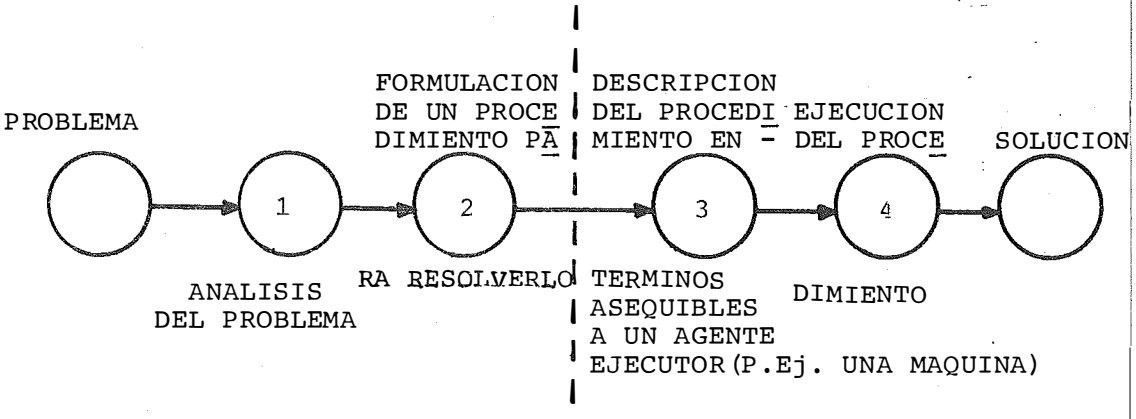


FIG. 1.

Si el problema tiene solución, la descripción del procedimiento para llegar a ésta estará fuertemente teñida por las características interpretativas y operativas del agente ejecutor del procedimiento que, en nuestro caso, será normalmente una máquina.

Así pues, el momento del proceso que se señala con una línea vertical de trazos marca normalmente la frontera a partir de la cual la principal preocupación del autor del algoritmo - es su comunicación con la máquina. Cuanto más evolucionada sea ésta menor será el esfuerzo del autor. De hecho, podemos esperar que la etapa 3 pueda desglosarse en una secuencia 3',3", - 3'", 3""..... tal que las reglas descriptivas en un lenguaje sean transformadas por una máquina M' en reglas descriptivas - en otro lenguaje, que sean transformadas por una máquina M" en reglas descriptivas en un nuevo lenguaje, etc.... hasta llegar a una máquina que sea capaz de ejecutar dichas reglas.

Por la misma razón, las etapas 2 y 3 pueden verse fundidas si se es capaz de formular el procedimiento directamente en términos de reglas o instrucciones para una máquina. En definitiva, la secuencia 3', 3", 3"',..... supone contar con la apoyatura de un encadenamiento de algoritmos de transformación de expresiones simbólicas, y, por consiguiente, la repetición otras tantas veces, y a niveles y para problemas distintos, - del proceso 1 - 2 - 3 - 4. Resumiendo, *el grado de concentración o multiplicación de las fases dependerá del conocimiento del sujeto acerca del problema a resolver y de la operatividad de los recursos ejecutores de que disponga.*

Fijémonos ahora en la ejecución del procedimiento o algoritmo, con independencia del problema y de la máquina de que se trate para lo cual daremos unas definiciones más formales - que en el apartado anterior.

2.1. El concepto de algoritmo, visto desde la teoría de conjuntos.

Se define un algoritmo como una cuadrupla A

$$A = \langle Q, E, S, F \rangle \qquad (1)$$

con Q, conjunto de todos los elementos simples y de todas las K-uplas que pueden describir el cálculo.

E, subconjunto de Q. Sus elementos son los datos de entrada al proceso de cálculo.

S, subconjunto de Q. Sus elementos son los diferentes resultados al término del cálculo.

F: $Q \to Q$, aplicación que describe la regla de cálculo propiamente dicha y que, a partir de cualquier elemento q_0, genera la construcción de una sucesión q_0, q_1, q_2.... tal que:

$$q_{i+1} = F(q_i);\ i \in N \text{ con } q_0 \in E \subset Q \qquad (2)$$

Para que A represente un algoritmo, cada sucesión (2) debe ser finita. Así pues, es preciso, como condición necesaria no suficiente, que la función F deje invariante el subconjunto S; es decir: $\forall\ s \in S,\ F(s) = s$. Si la sucesión es finita su punto de parada viene dado por el menor índice i para el que $q_i \in S$. No será finita si $\nexists\ i \in N;\ F(q_i) \in S$

Ejemplo: Cálculo del m.c.d. de dos números enteros no negativos n_1 y n_2. El organigrama de la figura 2 expresa un procedimiento conocido para resolver este problema. (P.E. significa Parte Entera de dividir n por n').

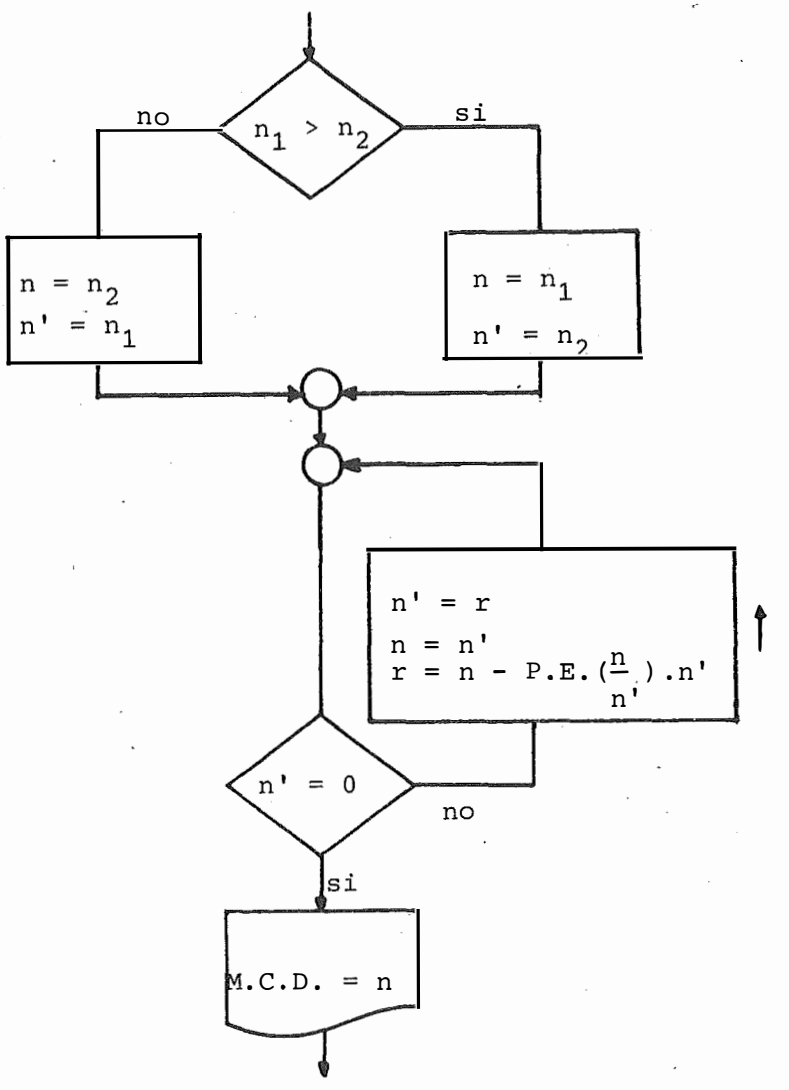


FIG.2.

Este algoritmo que se expresa en la figura 2 en una forma, en parte icónica (diagrama), en parte formal (expresiones matemáticas), puede representarse a través del lenguaje de los conjuntos así:

Q: (<n>, <n,n'> , <n,n',r,1>, <n,p,r,2>, <p,n',r,3>),

n,n',p,r ∈ N

E: (<n,n'>)

S: (<n>)

F: $F(n_1,n_2) = (n_1,n_2,0,1)$ si $n_1 > n_2$, si no

$F(n_1,n_2) = (n_2,n_1,0,1)$;

$F(n) = (n)$;

$F(n,n',r,1) = (n)$ si $n' = 0$, si no

$F(n,n',r,1) = (n,n',n-n' \times P.E.\ (n/n'),2)$;

$F(n,p,r,2) = (p,p,r,3)$;

$F(p,n',r,3) = (p,r,r,1)$;

El signo ; separa las distintas variantes de la regla de cálculo que el lector puede usar para hallar la sucesión $q_0,....q_i,...$ partiendo de cualquier pareja de números enteros no negativos, familiarizándose así con la definición que se acaba de dar. Por cierto que dicha definición habría que perfeccionarla en el sentido de que *la aplicación F no implique operaciones que no se sepa realizar*. En definitiva, las restricciones que habría que imponer a Q, E, S y F son de naturaleza tal que la cuadrupla A no contenga más que operaciones elementales simples (repare el lector en la relatividad del argumento de simplicidad, siendo la operación más simple la que pueda ejecutar un autómata).

Un ejemplo como el del cálculo del m.c.d. pone de manifiesto, mediante el empleo de los números 1,2,3, etc., la idea de que *una aplicación o un cálculo es una secuencia de aplicaciones o cálculos más elementales y que tal secuencia puede expresarse mediante un número asignado a las órdenes que deben ejecutarse.*

De forma general, podría decirse que el estado del algoritmo es un par (a,j), donde j indica el nº de la orden que debe ejecutarse y a es la información que caracteriza el estado del algoritmo cuando hay que ejecutar la orden j (información que comprende combinaciones de datos, resultados intermedios y resultados finales). Así, a cada evaluación de la aplicación $F(q_i) = F(a_i,k) = (a_{i+1},l)$ se le puede llamar ejecución de la orden k del algoritmo (Alabau, Figueras, 1.975). El conjunto S incluiría la información correspondiente a los resultados finales y el conjunto E, la información de los datos de entrada.

El concepto de algoritmo puede examinarse formalmente también en relación con un alfabeto y en términos de una máquina de Turing, cosa esta última que haremos más adelante. De momento, veamos más de cerca y en forma intuitiva las implicaciones de los algoritmos sobre las máquinas.

2.2. Las máquinas, como estructuras capaces de ejecutar algoritmos.

Toda máquina capaz de ejecutar algoritmos (Fase 4 del proceso de la figura 1) debe tener una estructura con los subsistemas que se destacan en la figura 3. (Alabau, Figueras, 1.975).

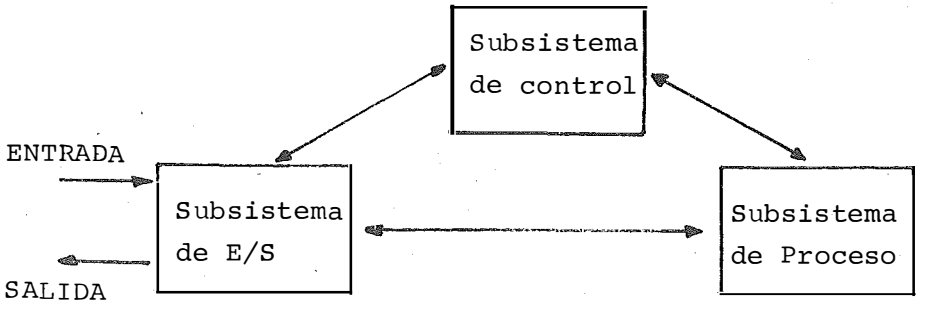


FIG. 3.

Siguiendo literalmente a estos autores, se definen a continuación los subsistemas básicos de una máquina general ejecutora de algoritmos y el subsistema de memoria que resulta de agrupar todos (o parte) de los elementos de memorización necesarios a la máquina.

2.2.1. Subsistema de entrada/salida, (E/S)

Donde se ejecutarán las órdenes de comunicación de la máquina y su mundo exterior.

2.2.2. Subsistema de proceso.

Donde se ejecutarán las órdenes de tratamiento de datos que dan lugar a la realización de operaciones del algoritmo.

2.2.3. Subsistema de control.

Donde se consigue el secuenciamiento adecuado en la ejecución de las órdenes y donde se generan las señales de control adecuadas para el funcionamiento de los subsistemas definidos en los dos apartados anteriores.

Por lo visto en el apartado 2.1, la ejecución de las órdenes de un algoritmo genera un nuevo estado q_{i+1} a partir del estado anterior q_i. Ello implica que hay que memorizar en la máquina el estado del algoritmo. Esto es lo mismo que decir que los subsistemas podrán llevar asociados elementos de memoria (asociados siempre, como se sabe, a todo circuito secuencial) para memorizar las informaciones de estado que corresponden a la misión de cada subsistema.

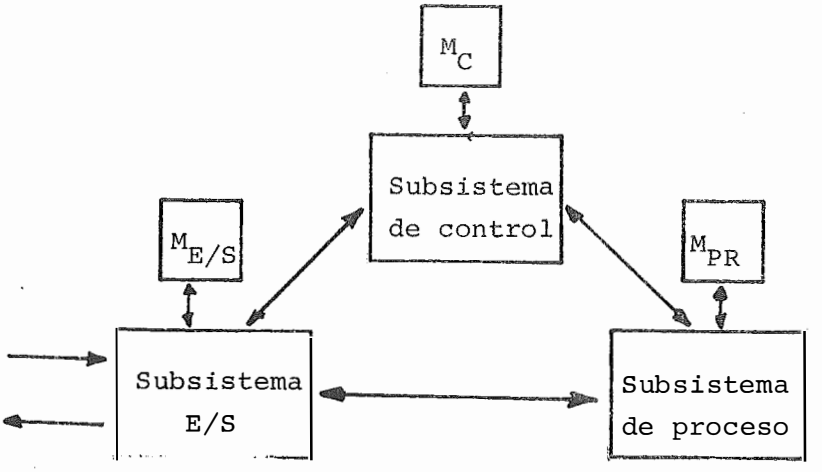


FIG. 4.

En muchas máquinas (y esto es precisamente lo que ocurre en los ordenadores) resulta más conveniente agrupar todos o una parte importante de los mencionados elementos de memorización en un subsistema específico, el subsistema de memoria - (figura 5).

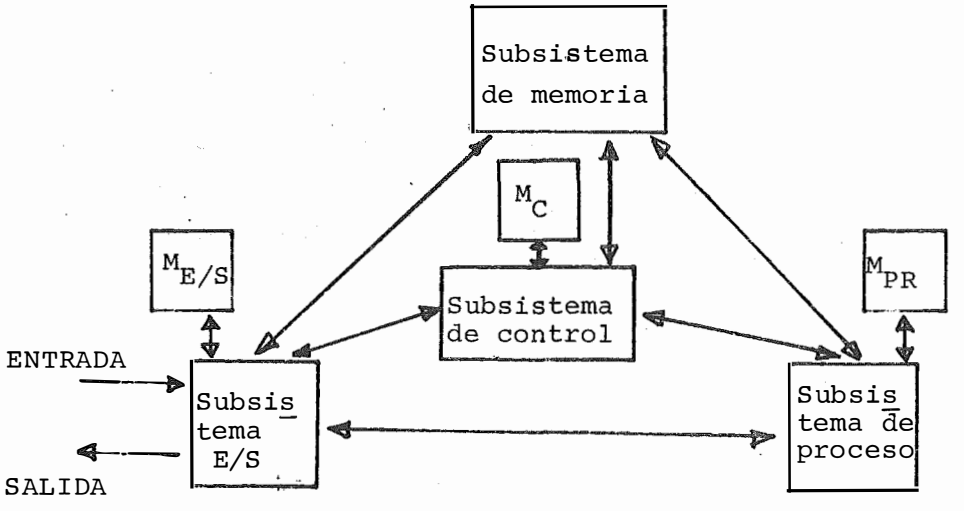


FIG. 5.

3. PROPIEDADES DE LOS ALGORITMOS.

Como resumen y ampliación de lo dicho describimos en este apartado las cuatro propiedades que debe poseer todo buen algoritmo (Corge, 1.975):

3.1. Propiedad de finitud.

Un algoritmo debe siempre terminarse. Todo algoritmo puede subdividirse en un número de subalgoritmos tan grande como se quiera, pero finito, admitiendo cada uno de estos subalgoritmos cadenas últimas que se terminan.

3.2. Propiedad de definitud.

Toda regla de un algoritmo debe definir perfectamente la acción a desarrollar, para aplicarla sin que pueda haber lugar a cualquier ambigüedad de interpretación.

3.3. Propiedad de generalidad.

Un algoritmo no debe contentarse con resolver un problema particular aislado sino, por el contrario, toda una *clase de problemas* para los que los datos de entrada y los resultados finales pertenecen respectivamente a conjuntos específicos.

3.4. Propiedad de eficacia.

Aun cuando un algoritmo posea las tres propiedades anteriores, se busca mejorarlo por razones de economía, de realizabilidad o de rapidez.

4. PROBLEMAS SIN ALGORITMO.

Los matemáticos han mostrado históricamente su deseo - de resolver tipos de problemas cada vez más generales. Este deseo, como se ha visto, inspira la propiedad de generalidad, - que es una propiedad relacionada con el grado de potencia de - un algoritmo.

Esto significa, así en abstracto, que sería preferible construir un algoritmo para hallar todas las raíces de una ecuación del tipo

$$a_n x^n + a_{n-1} x^{n-1} + ...+a_1 x + a_0 = 0$$

donde n es un entero positivo arbitrario, a construir un algoritmo para resolver la ecuación $x^n-a = 0$,o, más sencillo todavía, un algoritmo para hallar raíces cuadradas. Ante este planteamiento surge inmediatamente una restricción de orden pragmático en escoger el nivel de generalidad congruente con los propósitos de quien tenga que resolver un problema. También, como se verá, al elevar el nivel de generalidad puede penetrarse en un entorno donde no existan soluciones o, al menos, soluciones conocidas.

Ahora bien, cuando el propósito es de índole teórica, se puede, como señala Trakhtenbrot (Trakhtenbrot, 1.960) llegar a elevar dicho propósito hasta el nivel del sueño de Leibnitz que consistía en buscar un algoritmo para resolver cualquier - problema matemático. Refinado este enunciado dió en uno de los más famosos problemas de la lógica matemática, el problema de la deducción.

Es sabido que, utilizando símbolos, cualquier proposi-

ción de una teoría matemática puede escribirse mediante una fórmula y esta fórmula es una palabra definida en un alfabeto. Entonces, la derivación lógica de una proposición se convierte en una cadena de transformaciones de palabras (cálculo lógico). El problema de la deducción se puede formular así (Trakhtenbrot, 1.960):

Para dos palabras cualesquiera (fórmulas) R y S de un cálculo lógico, determinar si existe o no una cadena deductiva de R a S. (S es la proposición y R es la premisa).

Se supone que la solución es un algoritmo para resolver cualquier problema de este tipo. Dicho algoritmo daría un método general para resolver problemas en todas las teorías matemáticas que se construyen de forma axiomática. La validez de cualquier proposición S en tal teoría sólo significa que puede deducirse del sistema de axiomas. Después, la aplicación del algoritmo determinaría si la proposición S era válida o no. Además, si la proposición S fuese válida, entonces podríamos encontrar un encadenamiento deductivo correspondiente en el cálculo lógico y de ahí recuperar un encadenamiento de inferencias que probaría la proposición.

Hasta ahora no se ha encontrado tal algoritmo. De hecho, no se han encontrado algoritmos para problemas menos generales. El matemático alemán Hilbert presentó en 1.901, en un Congreso que se celebraba en París, una lista de 20 problemas no resueltos. El décimo problema se enunciaba de la forma siguiente: *Hallar un algoritmo para determinar si cualquier ecuación diofántica dada tiene una solución entera.*

Precisamente se conoce un algoritmo para resolver una ecuación diofántica con una incógnita: $a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0 = 0$, pero no cuando contiene varias incógnitas, como es el caso de la siguiente ecuación: $a^2+b^2-c^2 = 0$.

Otro problema que no tiene solución conocida es el de averiguar si dos sucesiones de ceros y unos están relacionadas. (H. Wang, 1.965):

Podemos formar una sucesión de ceros y unos que tenga «descendencia» mediante las siguientes reglas:
1. Si la sucesión tiene menos de tres símbolos, parar.
2. Si la sucesión comienza por 0, borrar los tres primeros símbolos y añadir 00 al final.
3. Si la sucesión comienza por 1, borrar los tres primeros símbolos y añadir 1101 al final.
¿Hay un algoritmo para determinar si una de las dos sucesiones dadas es descente de la otra?

011010001001	101110110011
01000100100	1101100111101
0010010000	110011111011101
001000000	011110111011101
00000000	11011101110100
0000000	1110111010011101
000000	01110100111011101
00000	1010011011101000
0000	0011011101001101
000	10110100110100
00	1101001101001101
(parar)	(descendencia continua)

Si prescindimos de los detalles operativos relacionados con la ejecución de los algoritmos (detalles de enorme importancia en la práctica) puede decirse: a) que *un problema tiene solución cuando se es capaz de demostrar formalmente que existe un método que genera siempre, a partir de todo elemento del conjunto E, un elemento del conjunto S (expresión (1)) en un número finito de pasos;* b) *No existe solución cuando se es capaz de demostrar formalmente la inexistencia de algún método que pueda generar un elemento de tal conjunto S en las mismas condiciones expresadas en a).* Y, por último, c) *no existe solución conocida cuando no se es capaz de demostrar una cosa o la otra.*

5. RESUMEN.

Para empezar se han definido verbalmente los algoritmos como un conjunto o lista de reglas, instrucciones o prescripciones que especifican a un agente ejecutor una secuencia finita de operaciones para la resolución de un problema. El -

problema ha de ser lo más general que sea posible, aunque esta condición es siempre bastante relativa.

Cuando se pretende resolver un problema, se desarrolla un proceso de varias etapas en las que normalmente unas son - más próximas a la estructura propia del problema (etapas de análisis y de formulación de un procedimiento de solución) y otras más decantadas del lado de las capacidades operativas del (o de los) agente ejecutor (en muchas ocasiones, una máquina y modernamente casi siempre un computador, al menos en nuestro caso). Este proceso resulta tanto más largo o complejo cuanto mayor - sea la distancia entre la complejidad del problema y la capacidad operativa del agente ejecutor disponible.

Si se hace caso omiso de este último factor, puede - - plantearse una definición formal de algoritmo como una cuadrupla $A = \langle Q, E, S, F \rangle$ donde la regla de cálculo $F: Q \rightarrow Q$ lleva al algoritmo desde un estado inicial ($\in E$) a un estado final ($\in S$) en un número finito de pasos, supuesto que se sepa realizar F con todos los elementos de Q.

Pensando ahora precisamente en términos del factor anteriormente descartado vemos que el uso y el concepto de estado del algoritmo, en donde puede distinguirse un n° de orden - de ejecución lleva a concebir un agente ejecutor-máquina como una estructura con tres o cuatro subsistemas (entrada/salida, proceso, control y memoria).

No se tiene un algoritmo si el procedimiento elaborado no cumple las condiciones de finitud y definitud, a las que - pueden añadirse, si se quiere hablar de "buenos algoritmos" - las de generalidad y eficacia, relacionadas con el grado de ambición o de optimización con que se enfoque la resolución de - un problema.

De todas formas hay problemas que no tienen solución o, al menos, solución conocida. Esta última parte del capítulo nos muestra cómo el grado de generalidad de un algoritmo tiene unos límites y nos pone delante de temas fundamentales de las matemáticas modernas.

CAPITULO 2

PROGRAMACION ESTRUCTURADA: CONCEPTOS TEORICOS

0. INTRODUCCIÓN

El tema de la programación estructurada se desarrolla en dos capítulos, el capítulo 2 dedicado a presentar los *conceptos más importantes de la teoría de la P.E.* (así la llamaremos frecuentemente en adelante) y el capítulo 3 a describir cómo aplicar en forma coherente dichos conceptos en un *proceso de diseño de programas*.

En el capítulo 2 se contesta a las preguntas siguientes: ¿Qué es un programa y cuál es su relación con los algoritmos?. ¿Qué es un ordinograma y cuál es su relación con los programas?. ¿Qué es un programa estructurado y si se puede y cómo se puede transformar un programa no estructurado en un programa estructurado?. Y por último ¿qué ventajas e inconvenientes comporta la P.E.?.

1. DEFINICIÓN FORMAL DE PROGRAMA PARA ORDENADOR.

Es una sucesión de reglas llamadas instrucciones que definen completamente un tratamiento para ser ejecutado en ordenador y que, en la ejecución, es la realización de un cierto algoritmo $A = <Q, E, S, F>$. Se puede representar por una cuadrupla

$$P = < X, x_0, Y, \psi >$$

con: X, conjunto de todos los estados del ordenador.

$x_0 \in X$, estado inicial de la máquina, programa cargado y datos iniciales.

Y, conjunto de los estados finales de la máquina después del cálculo.

φ, función de transición de un estado interno al estado interno siguiente.

P representa a A si existe una aplicación g de E en x_0, una aplicación h de X sobre Q aplicando Y en S y una función r aplicando X en N tal que:

- si e ∈ E produce el resultado s a partir de e si y solamente si existe un y en Y que pueda ser producido como resultado por P a partir de g(e) y tal que h (y) = s.
- si x ∈ X entonces F (h(x)) = h ($(\varphi^{r(x)}(x))$ donde la notación $\varphi^{r(x)}$ significa que la función φ debe ser iterada r(x) veces.

2. DIAGRAMAS DE FLUJO, ORGANIGRAMAS, ORDINOGRAMAS

La mayoría de los algoritmos se ejecutan modernamente con ordenador. Aunque un ordenador es en definitiva un autómata finito o, si se quiere, un conjunto de autómatas finitos interconectados, lo cierto es que sus usuarios no tienen normalmente conciencia de ello, limitándose a comunicarse con él por medio de lenguajes formales cuya estructura está más cerca del lenguaje matemático (por ejemplo) que del lenguaje del autómata.

Si bien es una práctica que está empezando a ser cuestionada desde hace algún tiempo, *es muy popular describir el algoritmo en un principio* (fase 3, fig. 1 cap. 1) *mediante un diagrama de flujo, organigrama u ordinograma*, que de todas estas maneras se viene designando. Desde la época de los primeros programadores, Condesa de Lovelace, Adela Goldstine, Grace Hopper, John Von Neumann, etc. se han utilizado los grafos para expresar el flujo de desarrollo de cálculos.

Sin embargo, hasta hace muy poco la utilización de estos grafos carecía de un soporte riguroso. Böhm y Jacopini, *en el año 1.966*, pusieron las primeras piedras de *la teoría de la programación estructurada*, que parte de la consideración del conjunto de bloques o diagramas de flujo como un lenguaje bidimensional de programación. Dijkstra, Manna, Wirth, Kosaraju y otros autores se han distinguido por sus aportaciones teóricas en este terreno y Warnier, por sus aportaciones prácticas especialmente en lo relativo a las aplicaciones informáticas de gestión.

En el resto del capítulo nos basaremos en (Böhm y Jacopini, 1.966), (Mills, 1.975) y (Tabourier et al., 1.975), sobre todo en esta última referencia.

2.1. Función, programa, función de programa.

Una *función* es un conjunto de pares ordenados (a,b) tales que $(a,b') \in f \wedge (a,b'') \in f \Rightarrow b' = b''$. Si $(a,b) \in f$, se puede escribir $b = f(a)$ siendo a un argumento, b un valor de f. El conjunto de todos los argumentos es el dominio de f y el de todos los valores es la imagen de f.

Un *programa* es un conjunto finito de funciones, llamadas instrucciones: cada instrucción opera sobre un dominio finito, contenido en un conjunto común D llamado espacio de los

datos. Estas funciones toman valores en Q = D x P, que podemos llamar espacio de estados. Así, una ejecución de programa es - una secuencia de estados

$q_i = (d_i, f_i)$, $i = 0,1,\ldots$ tal que $q_{i+1} = (f_i(d_i)$ para $i=0,1..$

q_0 es el valor inicial de la ejecución

q_n es el valor final de la ejecución, si la secuencia es finita.

Se llama función de programa [P] a

[P] = $\{(q_0, q_n)$, q_n es el valor final de P si el va-- lor inicial es $q_0\}$

El programa es, pues, una regla específica, pero no - única para calcular la función [P].

2.2. Normalización de organigramas.

Un programa P puede representarse por un grafo G=(P,U) en que el conjunto de los nodos es el conjunto de instruccio-- nes y donde el conjunto U de los arcos está definido por

$U = \{(f,g) \in PxP, \exists d \in \text{dominio } (f), \exists d' \in D, f(d)=(d',g)\}$

o, lo que es lo mismo, hay un arco de f a g si la instrucción g puede ejecutarse inmediatamente después de la instrucción f.

Este grafo puede adoptar en último extremo la forma de un *grafo orientado con tres tipos de nodos:*

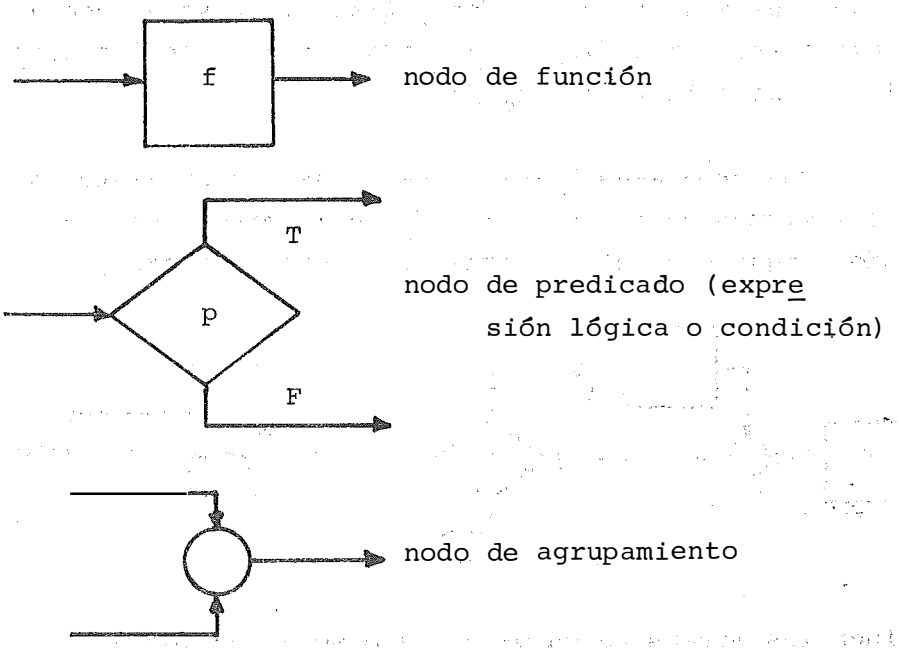


Fig. 1.

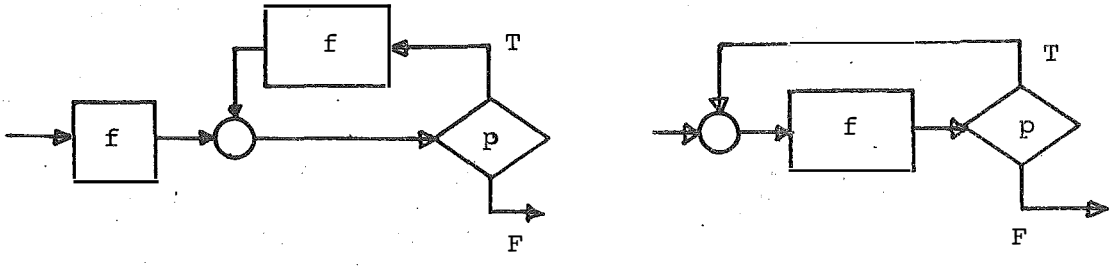
El nodo de función, en estas condiciones, conduce siempre a la misma instrucción siguiente, por lo que puede considerarse que la aplicación de su dominio no lo es en el espacio - Q = D x P, sino solamente en el espacio D.

El nodo de predicado tiene dos arcos de salida y es - una aplicación de su dominio en {True, False}({cierto, falso}). (El arco de arriba o de derecha, esto último si se traza verticalmente, corresponde a True, y el de abajo o izquierda corresponde a False). (Nota: a veces se cambia un lado por otro, en la práctica).

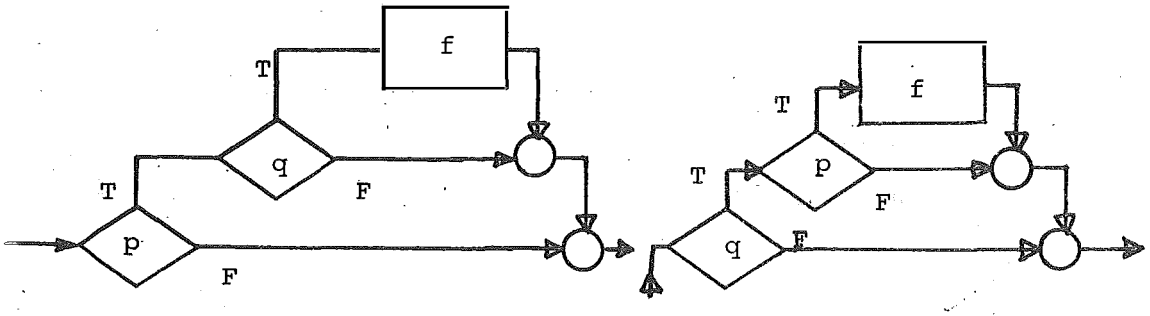
Un nodo de agrupamiento simplemente transfiere control de sus dos líneas de entrada a la de salida.

Definamos un *programa limpio* como aquel cuyo grafo o diagrama posee un solo arco de entrada y un solo arco de salida, existiendo un camino de entrada hasta cualquier nodo y de cualquier nodo hasta la salida.

Dos programas limpios pueden ser equivalentes si definen los mismos cálculos o si definen la misma función, aun cuando tengan distinto diagrama de flujo. Por ejemplo,



definen los mismos cálculos y las mismas funciones.

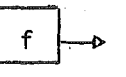


definen la misma función (cuando los arcos de salida de p y q son ambos T, entonces f en ambos grafos) pero diferentes cálculos (en los demás casos) (véase (Mills, 1.975)).

Por tanto, pueden definirse distintos niveles de equi

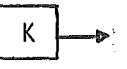
valencia, entre diagramas de flujo, unos que preservan cálcu-- los, otros que preservan función, etc.. Varios autores, ya men cionados, han estudiado la potencia de diversas clases de diagramas en la definición de cálculos y funciones. El principal resultado de sus estudios es que *clases relativamente pequeñas y económicas de diagramas de flujo pueden definir los cálculos y funciones de la clase de todos los diagramas de flujo, posiblemente a costa de cálculos extra fuera de la descripción original del conjunto de estados.*

Se define una clase de diagramas BJ (por Böhm y Jacopini) sobre un conjunto de funciones $F = \{f_1,...f_m\}$ y sobre un - conjunto de predicados $PR = \{P_1,...P_n\}$ así:

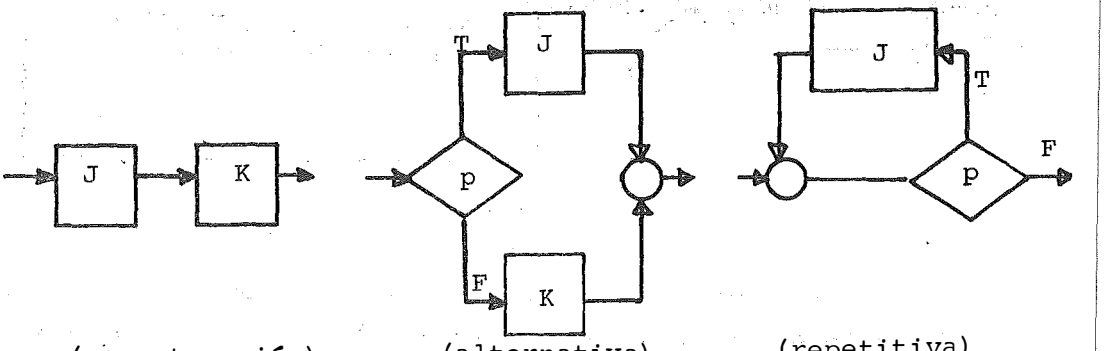
1. Si f ∈ F, entonces 

es un diagrama BJ

2. Si p ∈ PR y 

, 

son diagramas - BJ, entonces



(concatenación) (alternativa) (repetitiva)

son diagramas BJ.

La definición que se acaba de dar es recurrente y per mite construir cualquier diagrama de flujo de programas lim-- pios como un diagrama BJ. Esta es una forma de normalizar organigramas o diagramas de flujo. Damos a continuación dos teoremas (Mills, 1.975).

2.2.1. Un teorema de estructura.

Considérese cualquier diagrama de flujo cuyas funciones formen un conjunto F y cuyos predicados formen un conjunto PR. Aumentese los conjuntos F y PR con funciones y predicados que pongan y verifiquen variables fuera del conjunto de estados del diagrama dado. Existe, entonces, un diagrama BJ en los conjuntos ampliados que simula los cálculos del diagrama de partida.

En aplicación de este teorema, Cooper propone el siguiente método. Cójase cualquier organigrama y numérese cada uno de sus arcos. Entonces, el organigrama* que se establece más abajo, con una nueva variable L (de "Label"), simulará los cálculos del organigrama original.

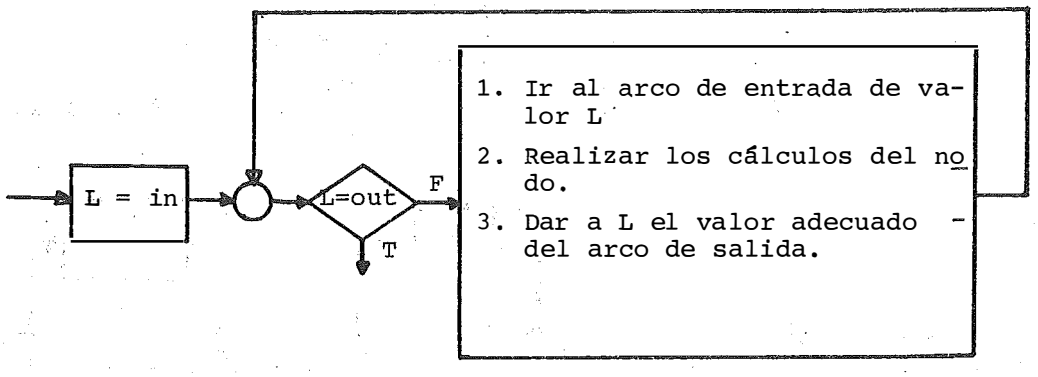


Fig. 2.

La operación dentro del bucle puede expandirse en un diagrama BJ sin iteración de tests sobre L, que conduzca a los distintos nodos del organigrama original como un conjunto de alternativas encajadas unas en otras. En resumen, este organigrama muestra que, a expensas de poner y verificar (hacer el test) de una sola variable L (fuera del conjunto original

* Observe el lector que, aún cuando las ramas T y F aparecen cambiadas con relación a la estructura repetitiva anteriormente definida, la estructura es BJ, idéntica a la que se obtendría negando el predicado y permutando el valor de las ramas.

de estados) pueden simularse los cálculos de cualquier organigrama como una subsecuencia de los cálculos de un diagrama BJ con un solo bucle (véase recuadrado un ejemplo de aplicación de este teorema sobre un organigrama que describe el flujo de operaciones para obtener el producto escalar de las filas i y j de una matriz de dimensiones (10 x 12) almacenada en unas determinadas posiciones del ordenador EIT-2; (tema 2, ejemplo 12, en el volumen I de esta misma obra).

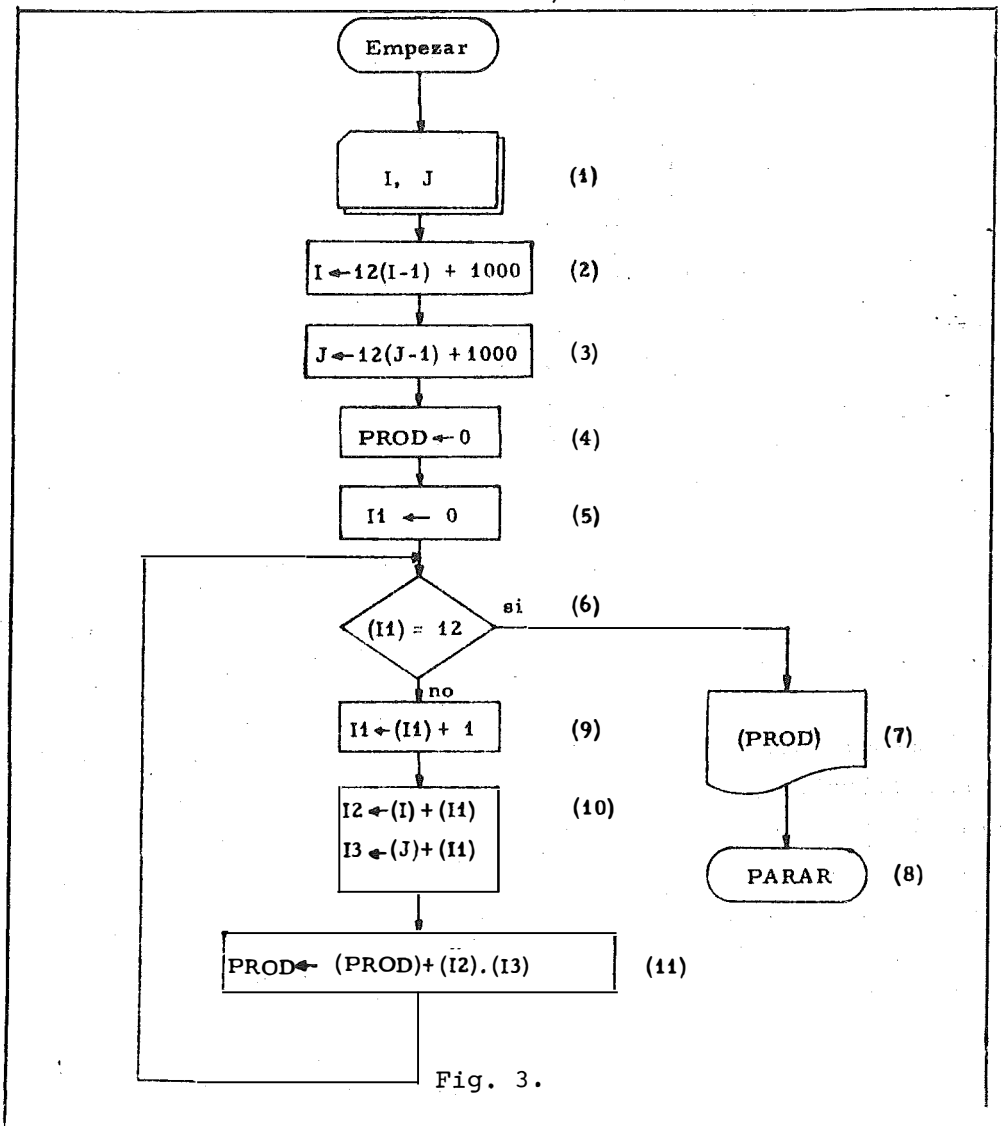


Fig. 3.

Si agrupamos y nos concentramos exclusivamente en la - estructura, el organigrama se sintetiza así (T = sí; F (de false) = no):

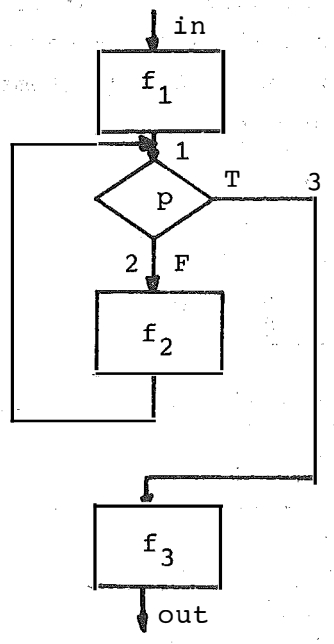


Fig. 4.

Aplicando el esquema y las ideas de la figura 2. Obtenemos el diagrama BJ de la figura 5.

A simple vista se aprecia que la figura 5, aún cuando simule los cálculos del diagrama de la figura 4 a través de un diagrama BJ, es mucho más pesada que ésta. Las estructuras son totalmente distintas. Y esto resulta especialmente paradójico en este caso particular, porque el organigrama de la figura 4 es de por sí casi un perfecto diagrama BJ, sin más que aplicarle ligerísimos retoques ($\neg$ significa la negación de la función):

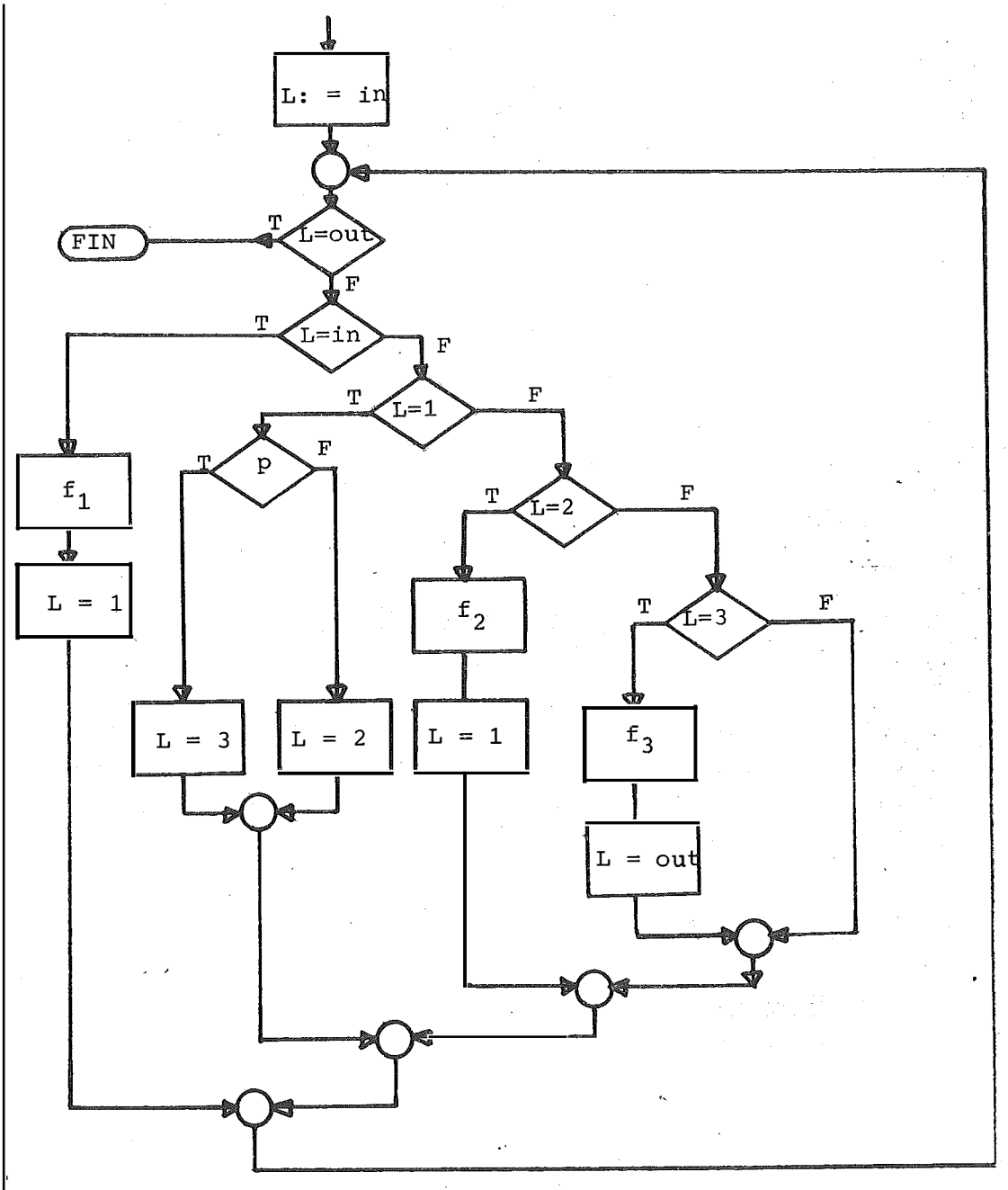

L: = in
FIN
T
L=out
F
T
L=in
F
T
L=1
F
T
p
F
f1
L = 1
T
L=2
F
f2
L = 3
L = 2
L = 1
T
L=3
F
f3
L = out

Fig. 5.

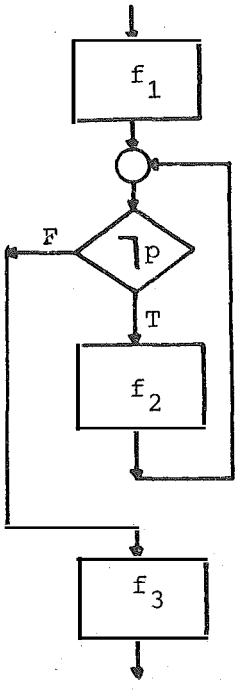


Fig. 6.

2.2.2. Teorema de estructura de Kosaraju.

Considérese cualquier diagrama de flujo G, cuyas funciones forman un conjunto F y cuyos predicados forman un conjunto PR. Existe, entonces, un diagrama BJ sobre F y P que conserva los cálculos de dicho organigrama G si, y sólo si, todo bucle de G tiene un solo arco de salida.

3. ¿QUÉ ES UN PROGRAMA ESTRUCTURADO?

3.1. Diagramas o estructuras básicas, fórmulas o esquemas de programa.

Los que hemos llamado diagramas BJ en el apartado 2.2. -algunos como (Mills, 1.975) les llaman diagramas D (por Dijkstra que es el autor que más y mejor ha escrito sobre estos temas)- son estructuras básicas porque se ha demostrado que con ellos es posible construir el grafo de cualquier programa. A - semejanza de lo que ocurre con cualquier expresión lógica y -

los operadores AND, OR y NOT. A veces es práctico utilizar dos estructuras más que resultan de una degradación o una reconstitución de las tres anteriores. En definitiva, *puede contarse con las cinco estructuras del siguiente cuadro, del que las estructuras 3 y 4 son derivadas de las tres primeras*. A menudo resulta cómodo designar estos grafos por una expresión o fórmula con un nombre. Los pioneros Böhm y Jacopini les llamaron π, Δ , Ω, Λ,y ϕ respectivamente por el orden en que los situamos

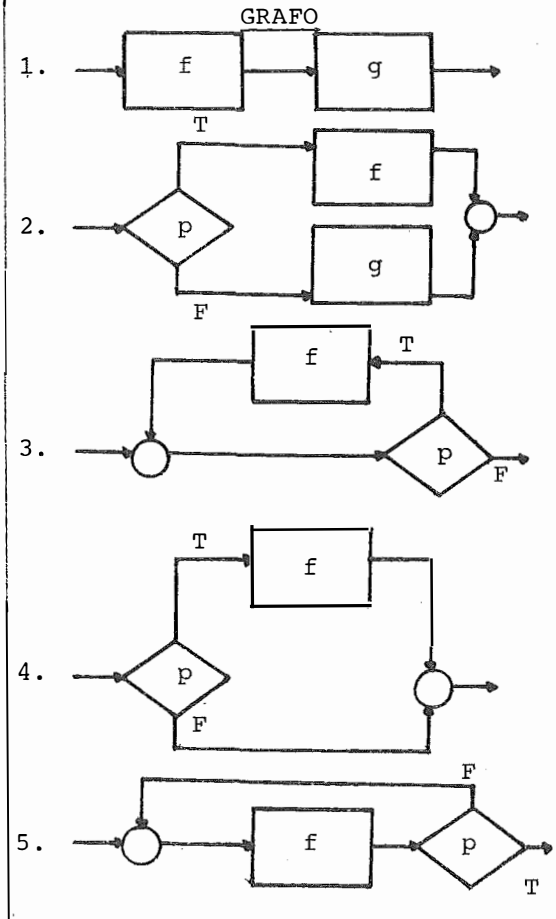


Fig. 7. ESTRUCTURAS BASICAS. (DIAGRAMAS Y FORMULAS)

en el cuadro. En la actualidad es más corriente denominarlos - por una expresión que se corresponde de cerca con algunos de - los lenguajes de alto nivel que describen tales estructuras.

3.2. Definición.

Se dice que un programa es estructurado si se expresa únicamente por medio de los diagramas del cuadro anterior. De manera más general, *un programa es estructurado si es una fórmula compuesta en cualquier conjunto prescrito de fórmulas básicas de programa.*

De manera más concreta, un programa es estructurado - (en el conjunto definido en 3.1) si tiene una de estas formas:

P = BLOCK (f,g)

P = IFTHENELSE (p,f,g)

P = IFTHEN (p,f)

P = DOWHILE (p,f)

P = DOUNTIL (p,f)

siendo p un predicado de P y f,g:

a) función identidad

b) cálculos de P

c) subprogramas limpios de P, también - estructurados.

El grafo de la figura 8 es estructurado, puesto que - puede escribirse:

P = IFTHENELSE (p, DOWHILE (q,f), BLOCK (g,h))

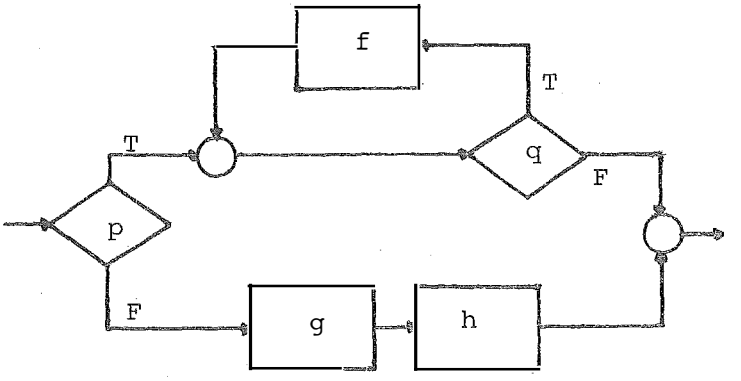


Fig. 8.

El grafo de la figura 6 es estructurado y puede escribirse así:

$$P = \text{BLOCK } (f_1, \text{ DOWHILE } (\neg p, f_2), f_3)$$

Al lector se le propone expresar la fórmula del diagrama de la figura 5.

4. TEOREMA DE ESTRUCTURA REFERIDO A UN PROGRAMA LIMPIO.

4.1. Funciones estándar y teorema.

Este teorema que vamos a enunciar sin demostración - - (véase en este sentido (Tabourier et al., 1.975) exige el uso de cuatro funciones estándar, TRUE, FALSE, POP y TOP. Aquí se presentarán estas cuatro funciones, el enunciado del teorema, una aplicación del mismo y un corolario.

TRUE: añade el valor "true" (cierto) al conjunto de los da-- tos que corresponden a su arco de entrada. Si a $\in$ A, siendo A el conjunto de datos, TRUE (a) = (a, true).

FALSE: añade el valor "false" (falso) a su dominio. FALSE - (a) = (a, false).

POP: se utiliza cuando una de las funciones TRUE o FALSE ha añadido una variable booleana al conjunto de los datos y su acción consiste en borrar esta variable. Si b es una variable booleana auxiliar, POP (a,b) = a

TOP: se utiliza cuando una de las funciones TRUE o FALSE ha añadido una variable booleana auxiliar al conjunto de los datos y su acción consiste en conservar el valor - de esta variable. TOP (a,b) = (a,b). Es un predicado. (b es la variable booleana).

Dice así el teorema: todo programa limpio es equivalente a un programa estructurado que contiene como máximo las fórmulas BLOCK, IFTHENELSE y DOWHILE (y/o DOUNTIL), las funciones estándar TRUE, FALSE, POP y TOP, así como las funciones y predicados del programa original.

Las funciones estándar hacen las veces de señalizado-- res para incorporar las estructuras básicas y guardar traza de la procedencia de las variables en el flujo de funciones.

4.2. Ejemplo de aplicación del teorema de estructura con - funciones estándar.

Se tiene el organigrama de la figura 9.

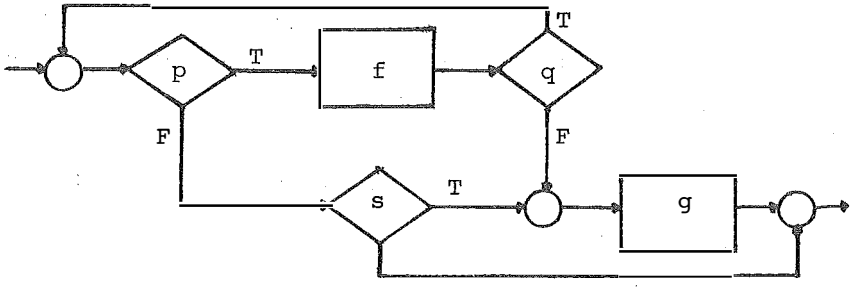


Fig. 9.

El organigrama de la figura 10 es equivalente al de la figura 9. Al comenzar la ejecución, se realiza la secuen--

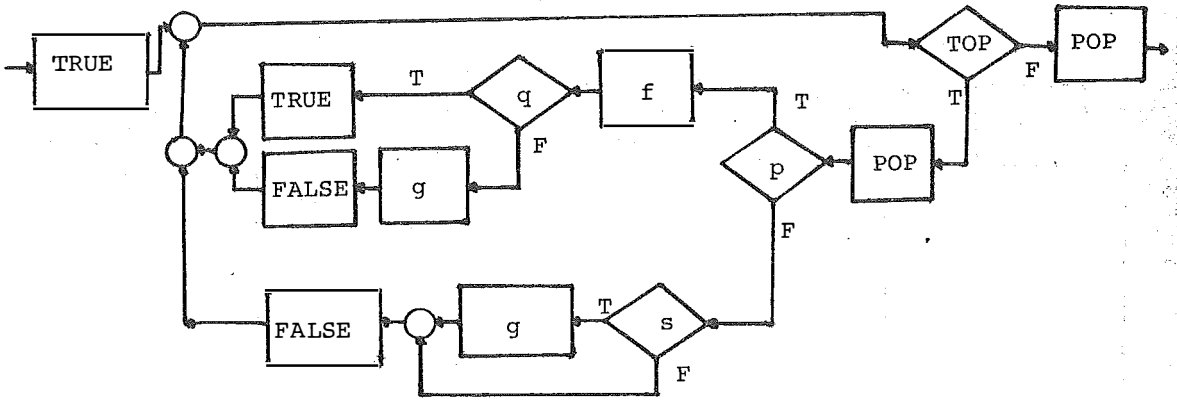


Fig. 10.

cia (TRUE; TOP; POP; p), por lo que la función p tiene como - dominio el conjunto de datos de partida, igual que en el programa original. En el ramal del grafo donde se encuentra f se marca uno de los arcos con TRUE y otro con FALSE para poder - encaminar de nuevo hacia la función p o hacia la salida, respectivamente. El ramal inferior, el de la secuencia (s;g) se marca con FALSE para poder encaminar el tratamiento hacia la salida por medio del mecanismo TOP.

El organigrama de la figura 10 es una estructura cuya fórmula puede escribirse de esta manera:

BLOCK (TRUE, DOWHILE (TOP, BLOCK (POP, IFTHENELSE (p,
BLOCK (f, IFTHENELSE (q, TRUE, BLOCK (g, FALSE))), BLOCK (IFTHEN (s,g), FALSE)))), POP).

El organigrama de la figura 11 es también equivalente al de la figura 9, aunque en esta ocasión se ha pivotado sobre la estructura DOUNTIL.*

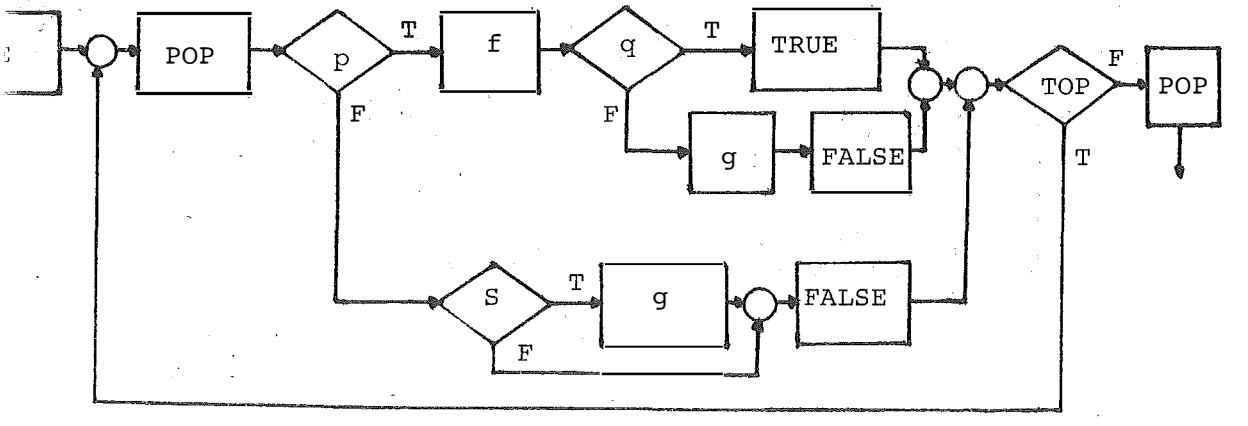


Fig. 11.

* Téngase presente la misma observación que la que se hizo para el esquema de la figura 2, válida allí para la estructura DOWHILE y aquí, por analogía, para la DOUNTIL.

4.3. Corolario

Todo programa limpio es equivalente a un programa en una de las formas:

BLOCK (f,g)

IFTHENELSE (p,f,g)

DOWHILE (p,f)

(y/o DOUNTIL (p,f))

donde p es un predicado del programa original o TOP, y donde f y g son programas limpios, cálculos del programa original, TRUE, FALSE o POP.

5. Interés y aplicabilidad de la Programación Estructurada.

La programación estructurada es, como se ha señalado, un movimiento de reciente aparición. Cabe indicar que su lanzamiento al gran público profesional se produjo prácticamente el año 1.973 en un número de la revista (Datamation (Datamation, Dic. 1.973) que dedicaba varios artículos al tema bajo el título genérico de *Revolución en la Programación.*

Coincide este lanzamiento con una etapa de análisis crítico de los problemas de la programación, relacionados primordialmente con la falta de fiabilidad de los programas, es decir del software en general, especialmente de los programas complejos. El software de base (sistema operativo, compiladores, etc...), por ejemplo, es un conjunto de programas muy complejos, y sus fallos conducen a consecuencias cuando menos desagradables y en muchos casos catastróficas.

La velocidad de expansión de la programación estructu

rada está condicionada al hecho histórico del momento de su - aparición, cuando ya un cierto número de hábitos y técnicas ha adquirido volumen. Hoy día incluso una persona formada en los principios de la P.E. y convencida de su utilidad tendrá que luchar contra el entorno. Casi cualquier libro o revista de informática que consulte le pondrá delante de los ojos un organigrama como el de la figura 12 (Madnick, Donovan, 1.974 pág.119) (Algoritmo de asignación de particiones en memoria) que no emplea los nodos del apartado 2.2, que no es un programa limpio ni mucho menos estructurado, aunque pueda ser un programa óptimo. En pocas palabras, no utiliza nada de lo que se ha estudiado en este capítulo. Este tema lo ha analizado F. Sáez Vacas - (F. Sáez Vacas, 1.975).

Al lector que por primera vez se encuentra con este - concepto a través de estas páginas le haremos unas anotaciones muy sumarias que le permitan orientarse en cuanto al interés y aplicabilidad de la P.E.

5.1. Comunicabilidad de la P.E.

Este es un aspecto que nadie puede discutir. La P.E. produce programas claros, expresados con un mínimo de fórmulas o diagramas, programas limpios. La consecuencia práctica inmediata es una disminución de los costes de programación relacionados con la tarea de modificación de programas y el trabajo en equipo.

Corolario de lo anterior es el favorecimiento del trabajo en equipo, cuyo ejemplo más notable es la técnica de organización del equipo estructurado de programación (*Chief Programmer Team*, CPT en la terminología profesional), que utiliza diseño modular y programación estructurada.

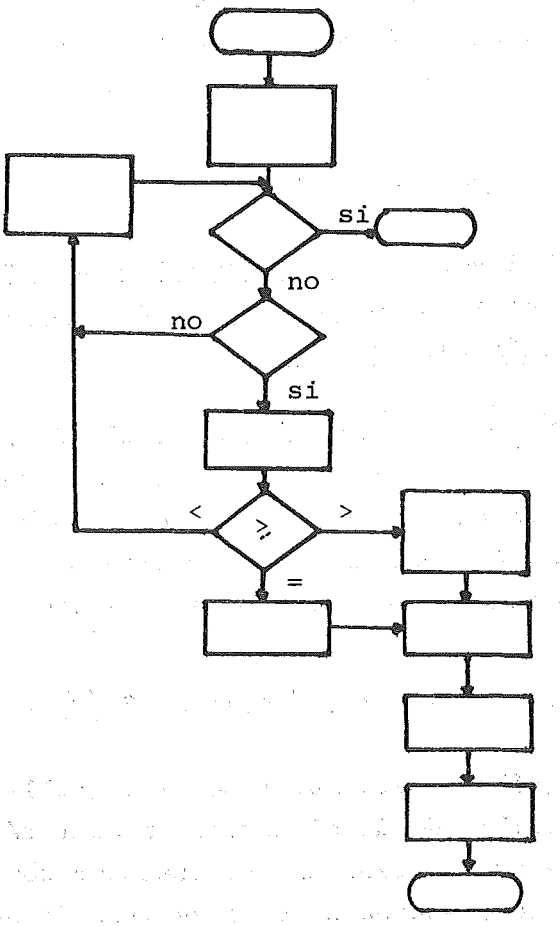


Fig. 12.

Otra faceta igualmente derivable es la didáctica. La P.E. puede enseñarse bien ya que, entre otras cosas, permite discutir y comparar programas.

En cuanto a la polémica "organigrama sí - organigrama no", la P.E. en un lenguaje estructurado de alto nivel (por - ejemplo Pascal, véase apartado 5.3) facilita el inclinarse - sin ningún problema por el abandono del organigrama.

5.2. Corrección de los programas.

Tal vez sea la posibilidad de demostrar formalmente si un programa es o no correcto el aspecto más importante de la - programación estructurada.

Efectivamente, si la fórmula de un programa emplea solamente los diagramas de la figura 7, puede probarse si aquella es o no es correcta mediante un censo de todos los nodos - del grafo. En cada nodo debe aportarse la prueba local de la - descomposición. Por ejemplo, si f = BLOCK (g,h), demostrar que

$$[f] = \{(r,t),\ \exists s,(r,s) \in [g] \wedge (s,t) \in [h]\}$$

(Véase, para mayor detalle (Mills, 1.975) y (Tabourier et al., 1.975) donde el lector encontrará nuevas referencias - si quiere profundizar en una cuestión como ésta que, siendo de gran interés, escapa a los objetivos de estas páginas).

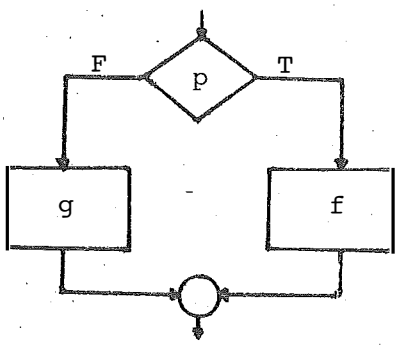
5.3. Los lenguajes de programación y la programación estructurada.

Los lenguajes más utilizados se han definido ajenos - por completo, y en general anteriormente, a la programación estructurada. Hubiera sido una casualidad que su estructura re--

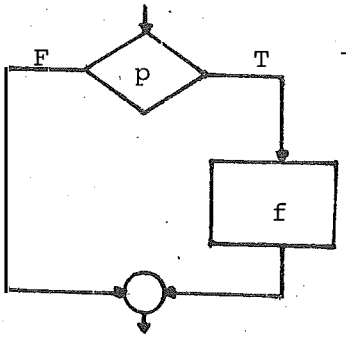
flejara precisamente las estructuras de la P.E. Y, en efecto, no la reflejan. He aquí un obstáculo objetivamente considera-- ble a la difusión de la P.E, producto del asincronismo en la - creación de dos tipos de lenguajes: el lenguaje gráfico formalizado (lenguaje de la P.E.) y el lenguaje formal lineal.

De los lenguajes formales existentes unos se adaptan - mejor que otros a la P.E, pero de manera genérica hay que aceptar ciertos retoques en la manera de utilizarlos, retoques que generan habitualmente una pérdida de eficacia en su uso. A título de ejemplo, el lector puede hacerse una idea (figura 13) de la clase de manipulaciones que habría que emplear en el uso de las sentencias de Fortran (Tenny, 1.974) para reflejar con fidelidad diagramas estructurados. Se pierde parte de la potencia del DO y sentencias tales como las de *IF aritmético*, *IF lógico*, *GOTO calculado y GOTO asignado*, por poner algunos ejem-- plos, sufrirían también serias erosiones al describirse diagramas estructurados mediante FORTRAN.

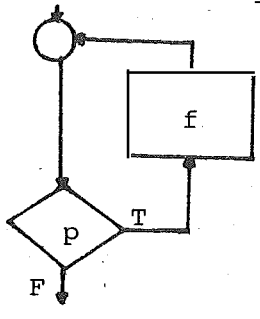
Existe un lenguaje,de creación muy reciente, denominado *Pascal* por su autor (Wirth), que está definido a imagen y - semejanza de la programación estructurada. Todavía está muy poco extendido. En 1976, el propio N.Wirth ha publicado un libro en el que todos los algoritmos se describen en dicho lenguaje y se han diseñado algunos compiladores de Pascal.



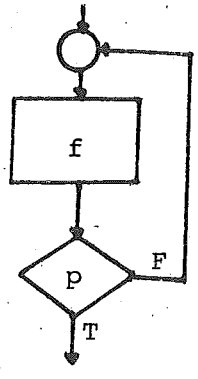

```
      IF (no p) GO TO 103
   f  {  ________
      {  ________
      {  ________
      {  GO TO  104
103
   g  {  ________
      {  ________
      {  ________
104   CONTINUE
```




```
      IF (no p) GO TO 102
   f  {  ________
      {  ________
      {  ________
      {  ________
102   CONTINUE
```




```
400   IF (no p) GO TO 401
   f  {  ________
      {  ________
      {  ________
      {  GO TO 400
401   CONTINUE
```




```
550   CONTINUE
   f  {  ________
      {  ________
      {  ________
      IF (no p) GO TO 550
```

Fig. 13.

6. RESUMEN.

Un programa es, mediando ciertas condiciones (véase apartado 1) *una representación de un algoritmo*. Análogamente, *un organigrama (ordinograma o diagrama de flujo) es un grafo representativo de un programa.*

Se ha expuesto cómo *un organigrama puede trazarse con sólo tres tipos de nodos (de función, de predicado y de agrupamiento)*. A su vez, *una sola clase de diagramas compuesta por tres diagramas simples (concatenación, alternativa, repetitiva) bastaría para definir los cálculos de la clase de todos los diagramas*. Estos diagramas simples poseen la característica de tener una entrada única y una salida única.

Se define como estructurado un programa cuando está formado exclusivamente con los diagramas o fórmulas de un determinado conjunto, como por ejemplo el constituido por {fórmula de concatenación, fórmula alternativa, fórmula repetitiva} . Naturalmente, el organigrama de un programa estructurado se expresa mediante una de estas tres fórmulas.

Parte del capítulo se ha dedicado a exponer y aplicar algunos teoremas que demuestran que programas no estructurados (limpios o no) pueden transformarse siempre en programas estructurados, generalmente a costa de romper la estructura del programa original, de ampliar el conjunto de estados del programa y de introducir nuevas funciones (apartado 2.2.1). Si el programa es limpio pero no estructurado, es factible llegar a un programa estructurado sin distorsionar excesivamente la estructura original introduciendo unas funciones concretas (TRUE, FALSE, POP, TOP), que amplían el conjunto de estados mediante la introducción de unas variables lógicas binarias.

La programación estructurada presenta grados diversos de interés y aplicabilidad, tema hoy muy discutido en el mundillo profesional. El apartado 5 recoge algunos rasgos de esta discusión, resaltando *la virtualidad teórica de poderse demostrar formalmente si un programa es o no es correcto*. Debe observar el lector que actualmente se prueba si los programas funcionan o no con un juego más o menos significativo del espacio de los datos, pero no se demuestra si son o no son correctos*.

---

* Hay una "ley" (Gilb, 1.975) en informática práctica que dice: *Todos los programas reales contienen errores mientras no se pruebe lo contrario, lo cual es imposible.*

CAPITULO 3.

PROGRAMACION ESTRUCTURADA: CONCEPTOS METODOLOGICOS

0. INTRODUCCIÓN.

En este capítulo nos proponemos describir, a través del desarrollo de un ejemplo muy sencillo, cómo se diseña un programa de nueva planta en forma estructurada. El método general consiste en manejar constantemente tres principios: a) el razonamiento será deductivo, de lo más general a lo más particular, b) a cada nivel de razonamiento se hará uso de los recursos abstractos necesarios, c) el razonamiento será guiado por la decisión de utilizar siempre estructuras básicas en la expresión gráfica de dicho razonamiento.

1. MÉTODO GENERAL DE DISEÑO DE PROGRAMAS ESTRUCTURADOS.

Sobre las estructuras básicas está casi todo dicho en el capítulo anterior, pero llamamos la atención del lector para que recuerde que la estructura completa de un programa estructurado tiene la forma de una estructura básica. De ahí partirá el método. Tras el análisis del problema se expresa un procedimiento de solución del mismo mediante un esquema estructurado muy general. Sin romper esta estructura, sino profundizando en su interior, se irá refinando el esquema mediante niveles o pasos sucesivos (stepwise refinement) hasta llegar a un nivel en que ya no interesa refinar más. Veamos qué son el "recurso abstracto" y el "razonamiento deductivo" (E. Reyero et al., 1.975):

1.1. El recurso abstracto.

Una de las circunstancias que más dificulta la concepción de programas reside en mantener simultáneamente "in mente" las especificaciones del programa, por un lado, y por otro lado los recursos concretos de que se dispone para codificarlas, es decir, las instrucciones del lenguaje de programación que se empleará. La tendencia tradicional consiste en plasmar directamente las especificaciones en términos de instrucciones.

Para salvar la brecha existente entre el dominio de las ideas tal como ocurren en la mente humana y el dominio de los procesos reflejados por las instrucciones de un lenguaje de programación, el diseño o concepción de programas estructurados se auxilia de lo que se denomina Recursos Abstractos por contraposición a los recursos concretos de que se dispone (un ordenador con un determinado lenguaje).

Según Dijkstra, (Dahl et al., 1.972) concebir un programa en términos de recursos abstractos consiste en descomponer una determinada acción compleja en términos de un número de acciones más simples, que podrían ser interpretadas como instrucciones para una supermáquina (inexistente) capaz de ejecutarlas. Puesto que nuestros recursos concretos son incapaces de procesar tales instrucciones concebidas para una máquina, el paso siguiente será, entonces, crear un programa que simule su funcionamiento, o lo que es lo mismo, descomponer cada una de sus instrucciones en acciones todavía más simples que podrían ser a su vez instrucciones para otra máquina ideal que plantearía los mismos problemas. El proceso continuaría hasta que en un determinado nivel de descomposición, las subacciones obtenidas constituyan instrucciones para el ordenador actualmente disponible.*

---

* En el capítulo 4 se verá hasta qué punto habría que llevar este proceso cuando el recurso concreto fuera una máquina de Turing.

Esta manera de concebir programas no es exclusiva de la programación estructurada. La programación modular la utiliza, no sistemáticamente, con otros fines. El programador experto que divide el programa en módulos hace algo parecido; los módulos serán codificados por distintos programadores, pero para él son como instrucciones de un lenguaje más potente que le permiten establecer más cómodamente el programa. Su problema, justamente es establecer qué módulos son los más adecuados, qué condiciones deben satisfacer y cómo deben ser encadenados. En realidad lo que establece son las especificaciones de la super-máquina que fuera capaz de procesar tales módulos, considerados como instrucciones. El resto de los programadores lo que hacen es simular la super-máquina con los recursos concretos disponibles.

Un aspecto que no debe ser olvidado en programación estructurada es el poseer unas estructuras las cuales son válidas para todas las super-máquinas; es decir, cuando una acción compleja se descompone en acciones más simples, la lógica con que se establecen tales acciones debe de corresponder con alguna de las estructuras. Este es el principio c) que se señalaba en la introducción a este capítulo.

1.2. Razonamiento deductivo ("top down").

Con este nombre se designa al proceso mental que permite concebir un programa por medio de una marcha analítica que se refleja en niveles o pasos consecutivos de refinamiento, cada uno de ellos poseyendo sus propios recursos abstractos que permiten resolver el programa por completo.

Esta marcha analítica, totalmente cartesiana, supone que situados en un determinado nivel de razonamiento, todos los niveles anteriores y cada uno de ellos han resuelto totalmente el problema; de lo que se trata es de hacer un refina-

miento que permita expresar esa solución de una manera más adecuada a ciertas limitaciones existentes (los recursos reales). Es por esto que NUNCA SERA NECESARIO VOLVER ATRAS* para reconsiderar ciertos aspectos no tenidos en cuenta, salvo error o - que se quiera intentar otra solución más conveniente.

El paso de un nivel al siguiente se hace por medio de un cambio en el "punto de vista", lo que permite introducir un refinamiento. Esto puede verse más claro reflexionando sobre - como es una estructura, habida cuenta de que solo tiene una entrada y una salida.

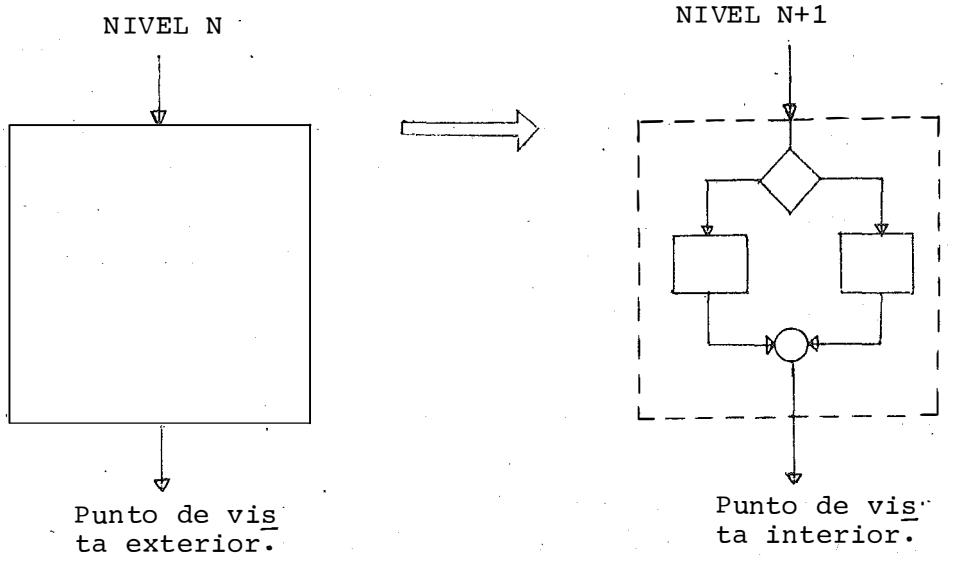


Fig. 1.

* En la práctica real, no se es capaz de realizar una marcha perfecta, sin vuelta atrás, pero éste es un principio que - es importante seguir con el máximo grado de pureza que sea posible.

Una estructura básica nos permite tomar dos puntos de vista con respecto a ella.

a) Punto de vista "exterior".

En este caso la estructura es vista como una caja "negra" con una única entrada y una única salida - por donde circula una información que la estructura manipula de alguna manera para entregar unos resultados en su salida.

El aspecto fundamental a considerar es que la estructura HACE algo, sin considerar "COMO LO HACE".

b) Punto de vista "interior"

Ahora la posición es totalmente inversa. Dada una - "caja negra" determinada que aporta una determinada acción, en un momento determinado debemos preguntarnos "de qué manera es hecho", si encadenando, ofreciendo alternativas o repitiendo acciones simples o complejas. Cuando tomamos esta posición, mirando a "COMO LO HACE" realmente, lo que tratamos es de averiguar la estructura interna de la caja negra primitiva.

1.3. Consejo al lector.

Al lector le habrá parecido este apartado tan incomprensible y "abstracto" como el recurso al que se ha hecho referencia. Es recomendable seguir con atención el ejemplo práctico del siguiente apartado, para releer posteriormente éste, que resultará entonces mucho más "concreto".

2. EJEMPLO DE DISEÑO DE UN PROGRAMA ESTRUCTURADO.

Se ha escogido un caso práctico que consiste en digitalizar el tiempo y en estructurar la programación de su simulación. El día 23 de Septiembre de 1.977 se propuso a los alumnos de la asignatura de Ordenadores Electrónicos de la E.T.S.-I.T. de Madrid resolver, entre otros, un problema (adaptado de (Arbib, 1.977)) cuyo enunciado reproducimos a continuación

2.1. Enunciado del problema.

Se pide simular mediante un programa escrito en ENSAM el funcionamiento de un reloj digital que posee dos ventanillas visualizadoras, la primera con tres campos, cada uno de dos dígitos decimales, que representan respectivamente de izquierda a derecha la hora, los minutos y los segundos (por ejemplo: 10: 22: 08 significa las diez horas, veintidos minutos, 8 segundos). La segunda ventanilla visualiza, mediante dos campos de dos dígitos decimales, el mes y el día del mes (p. ejemplo: 04: 30 quiere decir que nos encontramos en el mes de Abril, día 30).

El reloj funciona mediante un circuito integrado, compuesto, entre otros elementos, de tantos registros como campos de dígitos decimales hay en las ventanillas de la esfera del reloj. Dicho en otras palabras, a cada campo le corresponde un registro electrónico donde se crean las representaciones binarias como consecuencia de la recepción de los impulsos procedentes de un oscilador ajustado a un ritmo de segundos. A estos registros y a su simulación en ordenador les llamaremos HORA, MINUTO, SEGUNDO, MES, DIA.

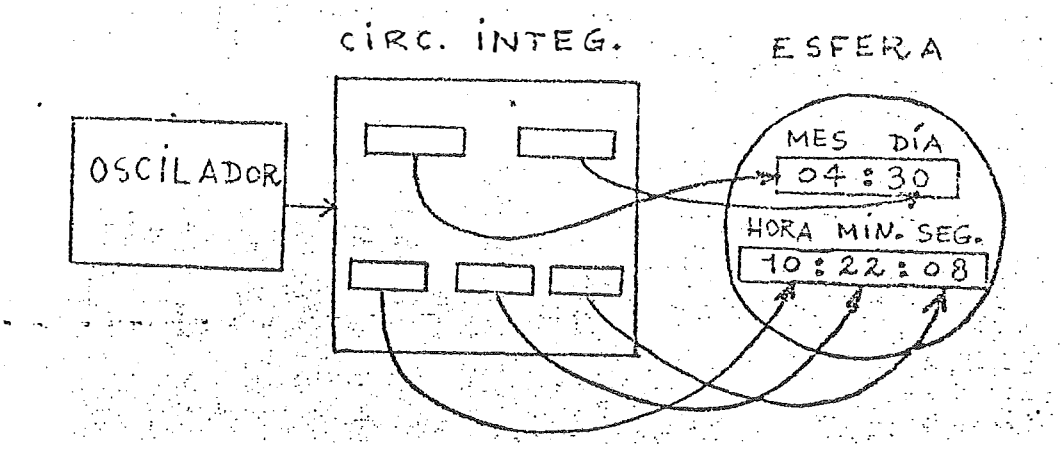


Fig. 2.

Suponemos que la batería que alimenta el circuito se agota al cabo de N impulsos, siendo N un dato numérico conocido. El programa, pues, leerá los datos N, HORA, MINUTO, SEGUNDO, MES y DIA, siendo estos cinco últimos los correspondientes al momento en que se pone en marcha el reloj. Se introducirá una variable de cuenta de impulsos llamada OSC.

Se pide que el programa escriba sucesivamente "HORA", N° DE HORA, "MINUTO", N° DE MINUTO, "SEGUNDO", N° DE SEGUNDO, "DIA", N° DE DIA, "MES", N° DE MES, una línea de seis asteriscos, "HORA", N° DE HORA, etc. Cuando se acabe la batería, el ordenador imprimirá "SE ACABO LA BATERIA".

OBSERVACIONES

RECORDAMOS que Febrero TIENE 28 DIAS (se despreciará la consideración de los años bisiestos), Septiembre, Abril, Junio y Noviembre tienen 30. El resto, 31

Obsérvese que se trata de obtener como salida los es-

tados sucesivos de la esfera del reloj, pero no hay que preocu parse de que la impresora escriba estas salidas a intervalos - de tiempo fijo.

2.2. ¿Por qué este ejemplo?

Este es un ejemplo que, en algún sentido, es muy ade-- cuado para los propósitos didácticos que animan este texto. En primer lugar, es sencillo intrinsecamente y permite poner de - manifiesto con claridad, en el espacio de unos 45 minutos que viene a durar una clase, los principios básicos del diseño de programas estructurados. De otra parte, se trata de un proble- ma conocido del lector y, por tanto, la dificultad de comprensión de su contenido no podrá ser una nube perturbadora que le impida concentrarse en lo esencial del ejemplo, que no es el - ejemplo en sí, sino el método que se va a seguir con él.

Sin embargo, en otro sentido es un ejemplo deficiente. Su misma sencillez animará al lector a creer que puede saltar- se pasos del método y, lo que tal vez sea peor, a hacerle pensar que no aparenta ser tan ventajosa la P.E. como se pretende. Ciertamente, los beneficios de la P.E. se aprecian con mayor - nitidez enfrentándola a problemas complejos.

2.3. Una solución no estructurada.

Aquí sólo interesa estudiar el organigrama que, como - se ha repetido, no es único. La codificación en un lenguaje, - sea el que sea (ENSAM, FORTRAN, ALGOL, etc.) no entra en nuestros objetivos ahora.

Para que el lector pueda establecer comparaciones es-- tructurales reproducimos en la figura 3 un organigrama tal como ha sido diseñado por una persona que ha estudiado, codifica

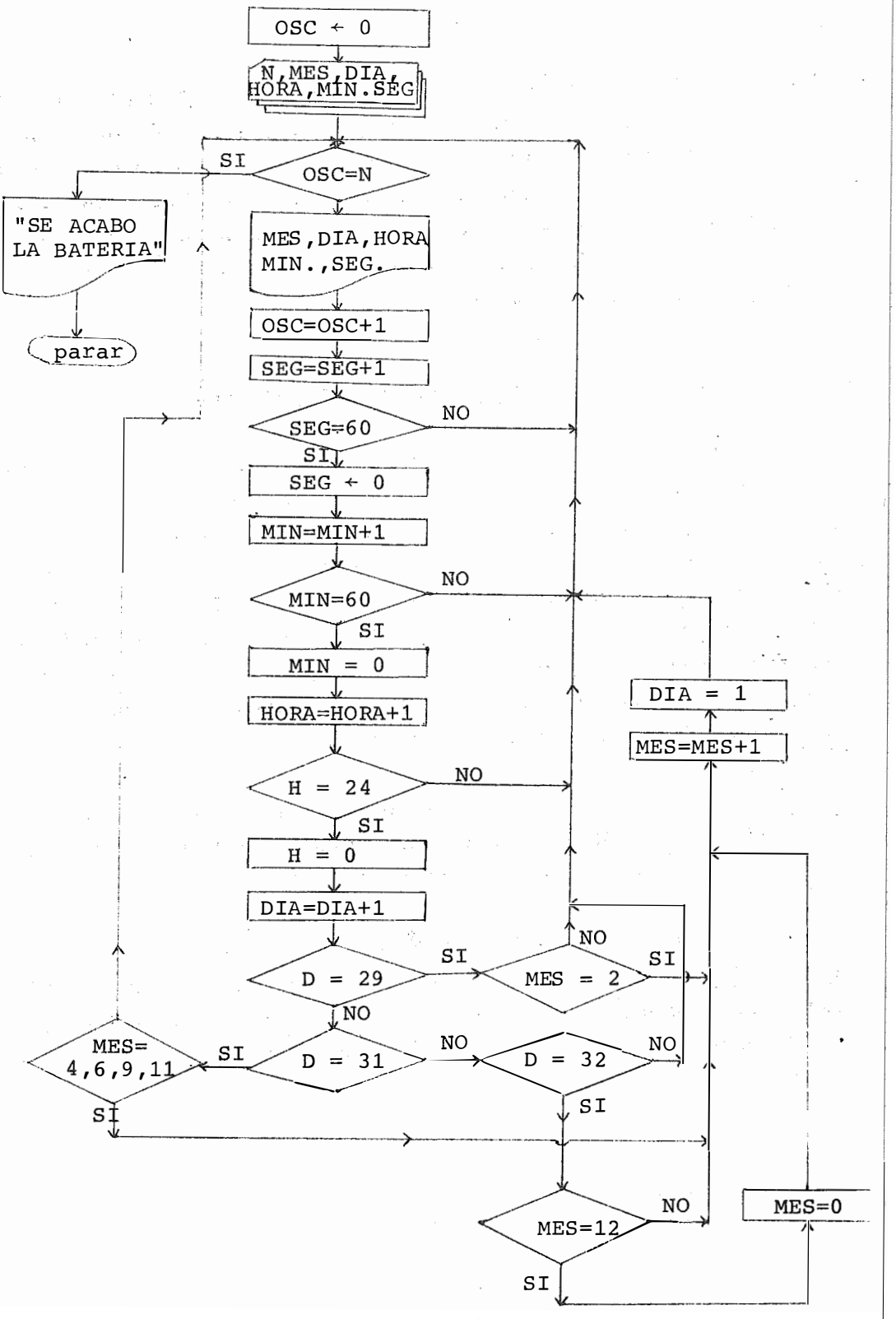

OSC ← 0
N,MES,DIA,
HORA,MIN.SEG
SI
OSC=N
"SE ACABO
LA BATERIA"
parar
MES,DIA,HORA
MIN.,SEG.
OSC=OSC+1
SEG=SEG+1
SEG=60
NO
SI
SEG ← 0
MIN=MIN+1
MIN=60
NO
SI
MIN = 0
HORA=HORA+1
H = 24
NO
SI
H = 0
DIA=DIA+1
DIA = 1
MES=MES+1
D = 29
SI
MES = 2
NO
SI
NO
MES=
4,6,9,11
SI
D = 31
NO
D = 32
NO
SI
SI
MES=12
NO
SI
MES=0

Fig. 3.

do y probado este programa. (Piénsese que un programa estructurado no tiene necesariamente por qué ser óptimo desde un punto de vista de minimización del tiempo de ejecución o de la ocupación de memoria. La programación estructurada busca la optimización en otros terrenos: en la minimización del tiempo de diseño o modificación de programas, en el rigor y fiabilidad de éstos, en su transferibilidad, etc.).

2.4. Desarrollo de una solución estructurada.

En el primer nivel se plantea una solución esquemática completa que resuelve totalmente el problema, en el caso de que fuera posible contar con una máquina que tuviera la capacidad de interpretar y ejecutar las instrucciones contenidas en los nodos del organigrama de la figura 4.1. Este organigrama tiene una estructura definitiva. Examinemos si algún nodo necesita un desarrollo más detallado, porque cuando así sea (y lo recuadraremos con trazo más fuerte) se tratará de una instrucción dirigida a un recurso abstracto.

En el esquema se entiende por HORA y FECHA los conjuntos (SEG, MIN, HORA) y (DIA, MES). Cuando HORA no aparece subrayado es que se refiere al elemento HORA.

Si se exceptúa la instrucción "CALCULA E IMPRIME NUEVA HORA y FECHA" las demás son prácticamente órdenes traducibles muy sencillamente al lenguaje de cualquier ordenador. Así, pues, se desarrolla dicha instrucción en un nivel de desglose superior (Fig. 4.2).

La instrucción "calcula hora y fecha" se va refinando sucesivamente, de manera que cada vez son menos las cosas que se piden a un recurso abstracto (figs. 4.3, 4.4, y 4.5). Al llegar al nivel representado en la figura 4.6 nos encontramos

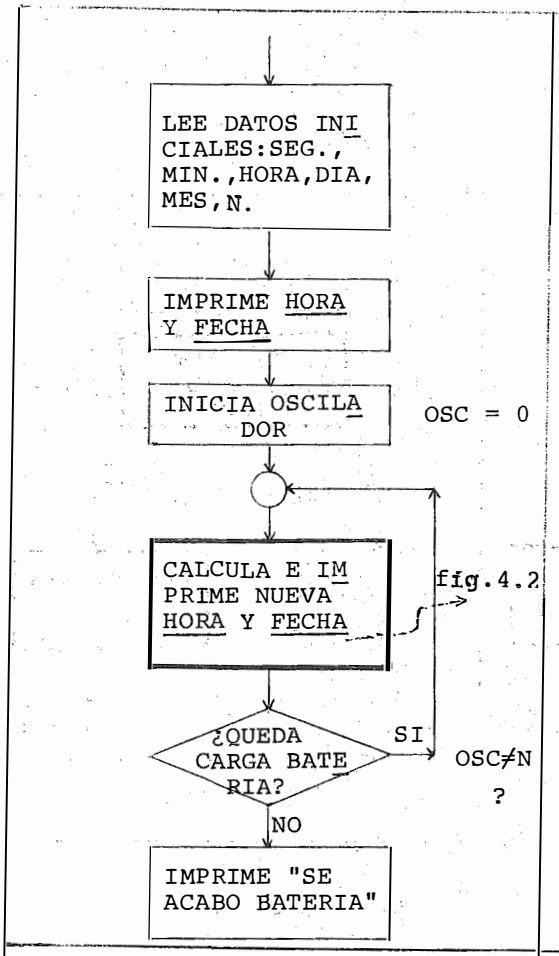


Fig. 4.1.

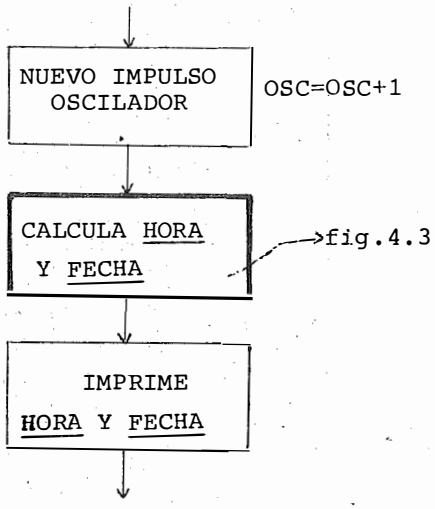


Fig. 4.2

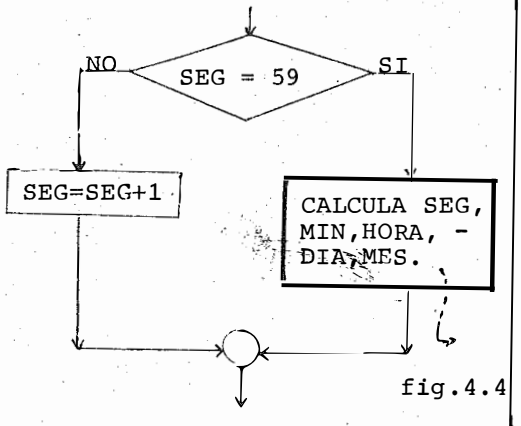


Fig. 4.3.

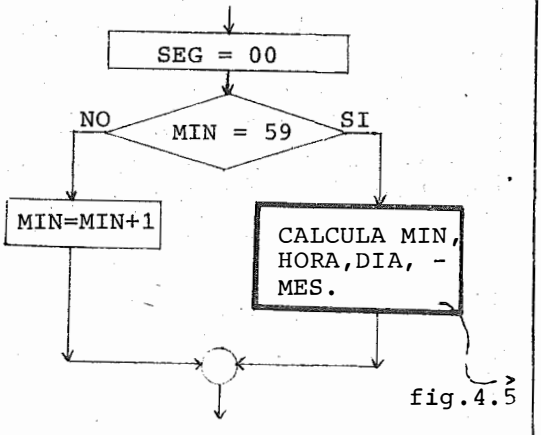


Fig. 4.4.

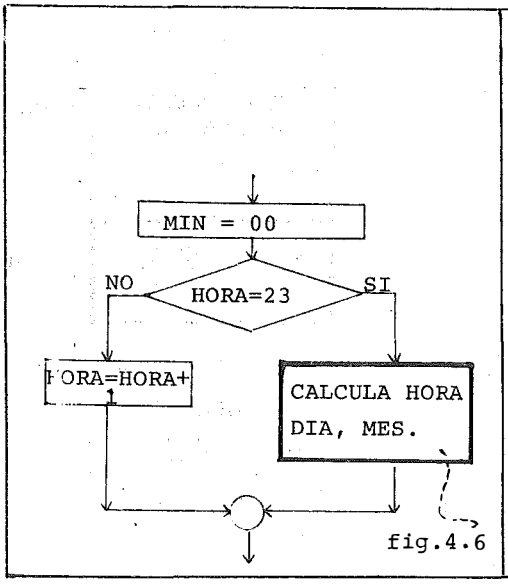


Fig. 4.5.

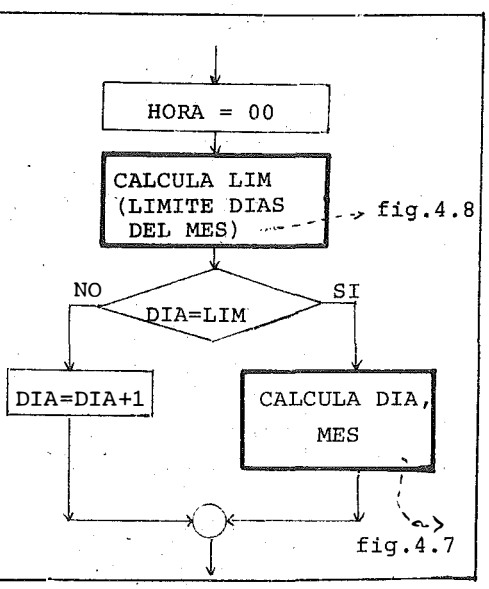


Fig. 4.6.

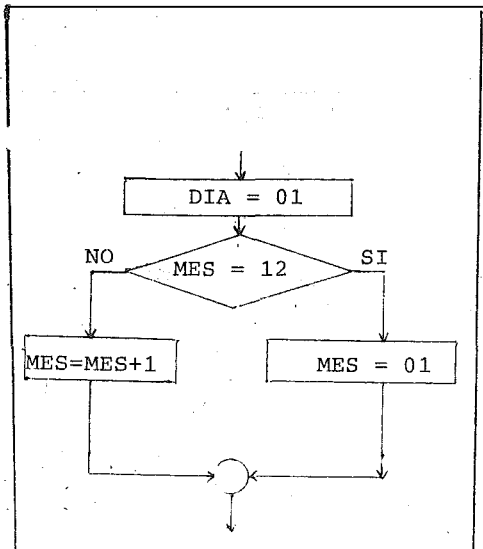


Fig. 4.7.

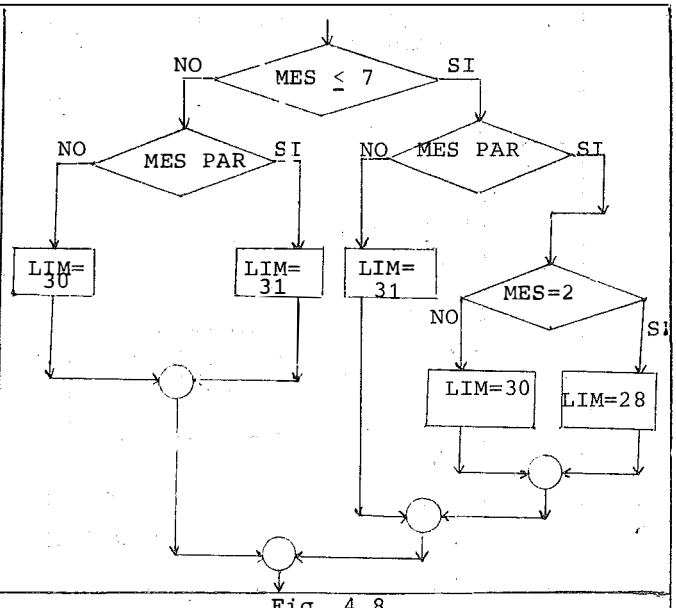


Fig. 4.8.

ante un pequeño obstáculo: todos los meses no tienen el mismo número de días. Este es un obstáculo normal. Lo que estaba ocurriendo hasta aquí no dejaba de ser una aburrida monotonía. Si el número de horas que constituyen un día dependiera del día de la semana o el de minutos que forman una hora de que fuera antes o después de mediodía, posiblemente nos haríamos un taco en la vida corriente, pero en lo tocante a la programación actuaríamos como en la figura 4.6. Se especificaría una instrucción a un recurso abstracto para que calculase el límite de la cuenta correspondiente a las circunstancias concretas. Y este desarrollo se dejaría para más adelante, en este caso para la figura 4.8, donde se plantea una solución estructurada entre varias posibles.

Una vez terminado el proceso de desglose, se recorre el camino en sentido inverso sustituyendo sistemáticamente los diagramas más sencillos en el diagrama de nivel inmediatamente anterior hasta recuperar la estructura completa de la figura 4.1, pero ya compuesta con todo el detalle de las operaciones fácilmente traducibles a instrucciones para un ordenador. El organigrama final es el de la figura 5, en donde por razones de falta de espacio no se ha sustituido el diagrama de la figura 4.8. (Es obvio, -el lector puede comprobarlo fácilmente- que este diagrama dista de ser el mejor. Así, sin merma de la buena estructuración del programa podría haberse introducido el cálculo del "límite de los días del mes" al iniciarse cada nuevo mes, siempre que se hubiera introducido como dato el límite del mes en el que se inicializa el reloj).

3. OBSERVACIONES SOBRE EL MÉTODO

Alcanzar un cierto dominio en la aplicación de las reglas generales del método de diseño que se acaba de ver es algo que se consigue con la práctica y no está exento de dificul

tades. Pero se hace necesario tomar siempre en cuenta que estas reglas no excluyen -sino más bien al contrario- el uso de la lógica que, en el caso de la programación, se vincula estrechamente a la estructura de los datos que ha de tratar el programa. El mismo ejemplo del reloj digital puede proporcionarnos la oportunidad de mostrar hasta donde nos llevaría el uso ciego (sin lógica) del método.

Supongamos que se nos pide modificar el organigrama de la figura 5 para simular el funcionamiento del reloj en el caso de que éste contase con una ventana visualizadora más, de dos campos, uno para el día de la semana (DIASEM) y otro para el día del año (DIAAÑO). (El primer día de la semana es el lunes y el año será siempre de 365 días).

El lector debe intentarlo por sí mismo, antes de mirar una solución que se le ofrece más abajo. Ciertamente el problema parece simple, y lo es. Sin embargo, un elevado porcentaje de los alumnos que tuvieron que resolverlo de manera imperativa (examen 13/5/78 en la E.T.S.I.T.M.) lo hicieron mal. Más o menos, como se indica en las figuras 4'.5, 4'.6, 4'.7 y 4'.8. En ellas parece haberse aplicado sistemáticamente (mejor diría se ciegamente) el método, a imagen y semejanza del proceso seguido en las figuras 4.1 a 4.8. El resultado es que el programa modifica el "día del año" sólo cuando cae en lunes, y el "día del mes" y el "mes" sólo cuando, además de caer en lunes fuera el primer día de un nuevo año.

¿Qué ha ocurrido?. Simplemente que no se ha tenido en cuenta que las variables DIASEM, DIAAÑO y DIA corresponden a tres enumeraciones distintas, sin relación entre sí, y que, por tanto, hay que llevar de manera disjunta. Véase una posible solución en la figura 4".6, que sustituiría a la figura 4.6. (Naturalmente, sería imprescindible leer DIASEM y DIAAÑO al principio del programa e imprimirlo a lo último).

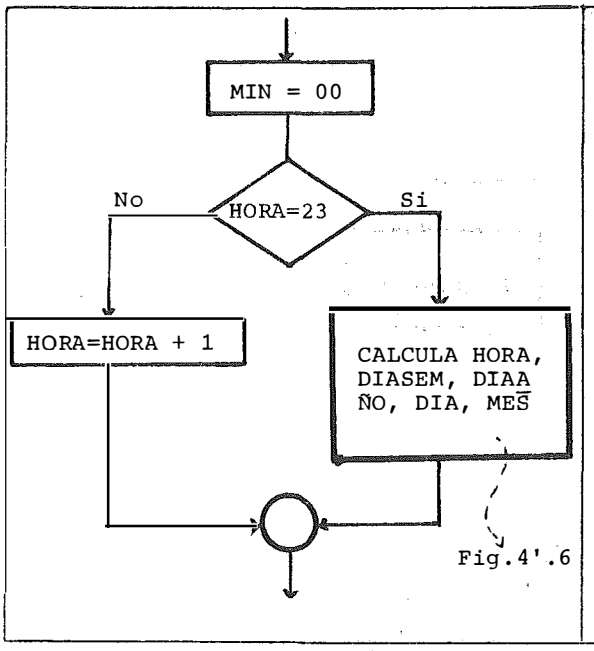


Fig.4'.5

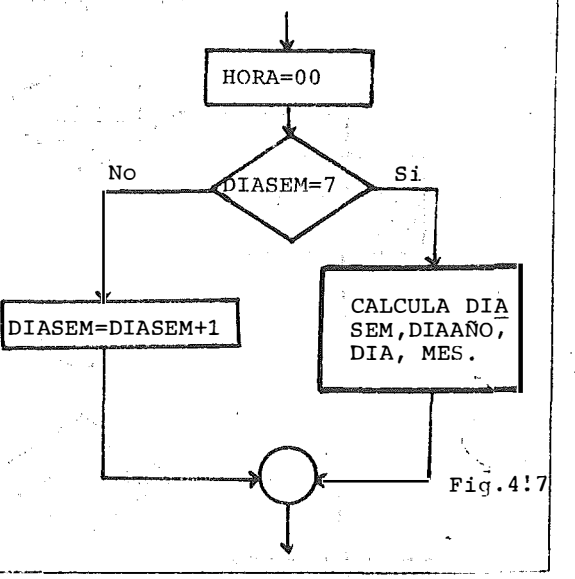


Fig. 4'.6

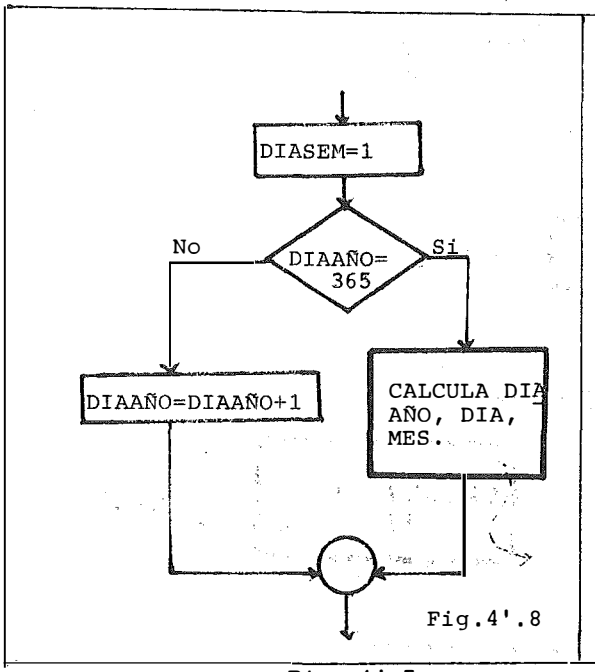


Fig. 4'.7

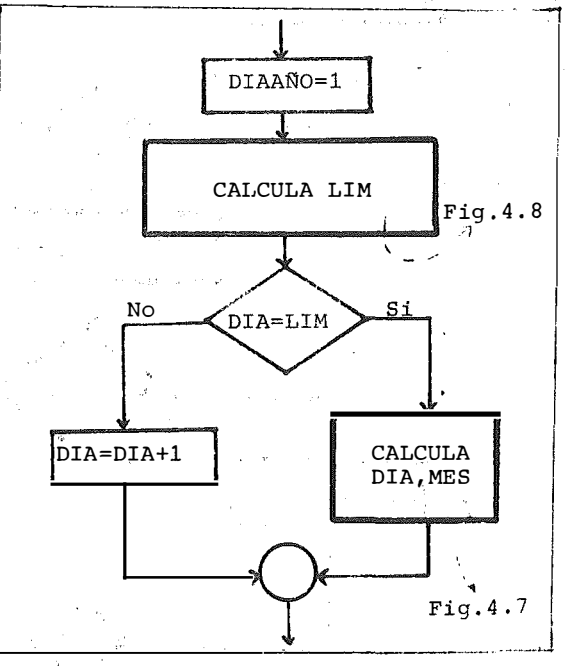


Fig. 4'.8

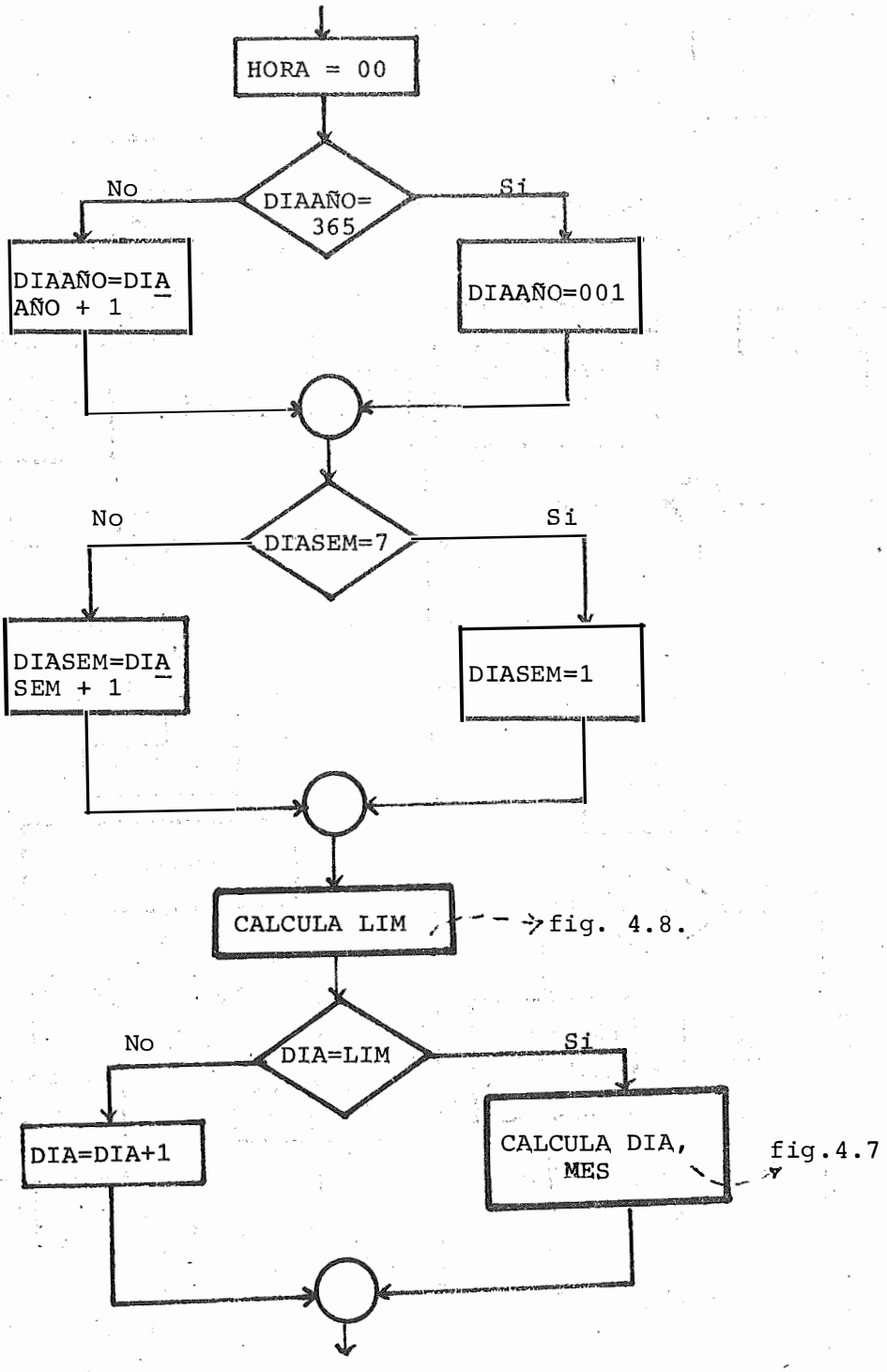

HORA = 00
DIAAÑO= 365
No
Si
DIAAÑO=DIA AÑO + 1
DIAAÑO=001
DIASEM=7
No
Si
DIASEM=DIA SEM + 1
DIASEM=1
CALCULA LIM
fig. 4.8.
DIA=LIM
No
Si
DIA=DIA+1
CALCULA DIA, MES
fig.4.7

Fig. 4".6.

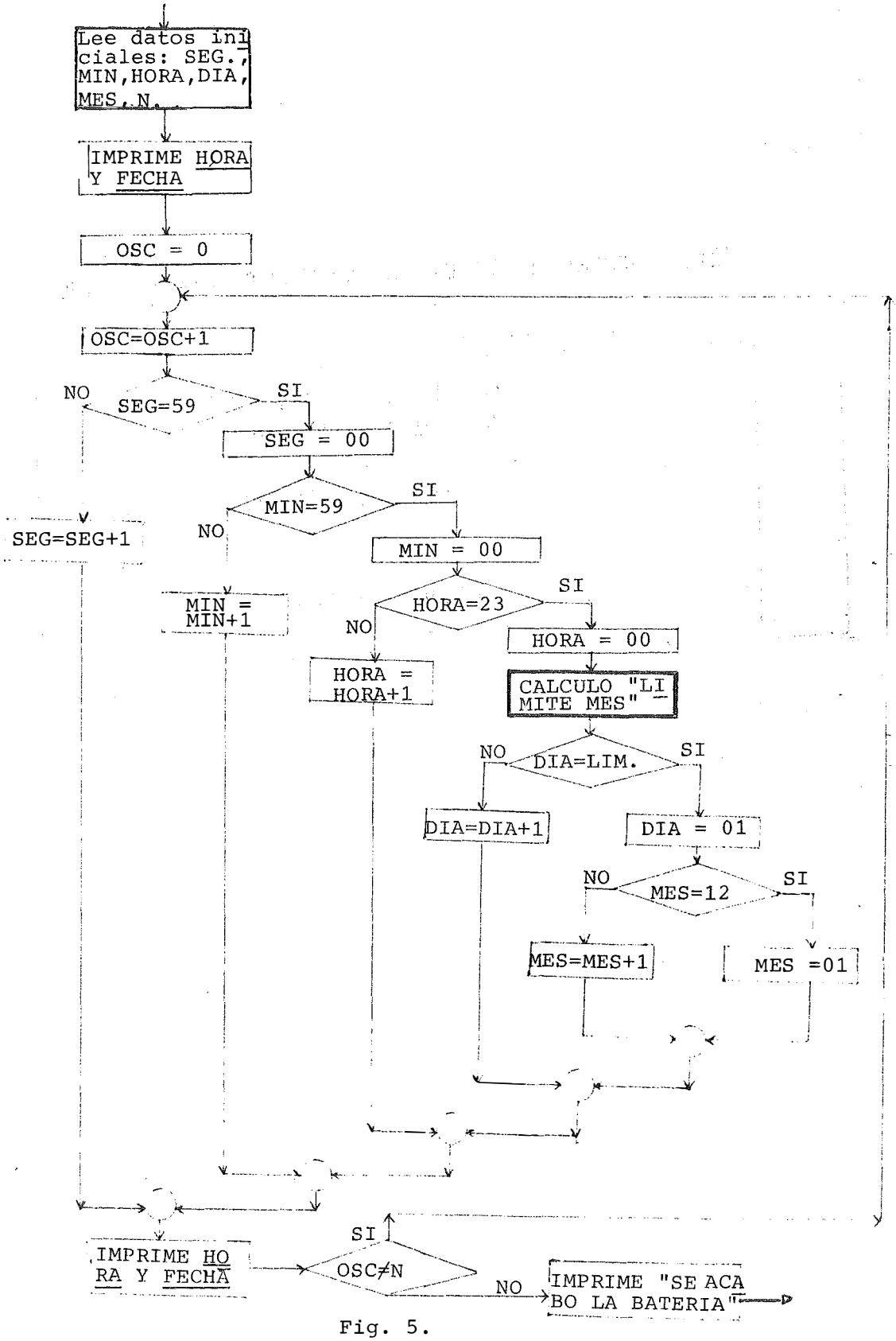
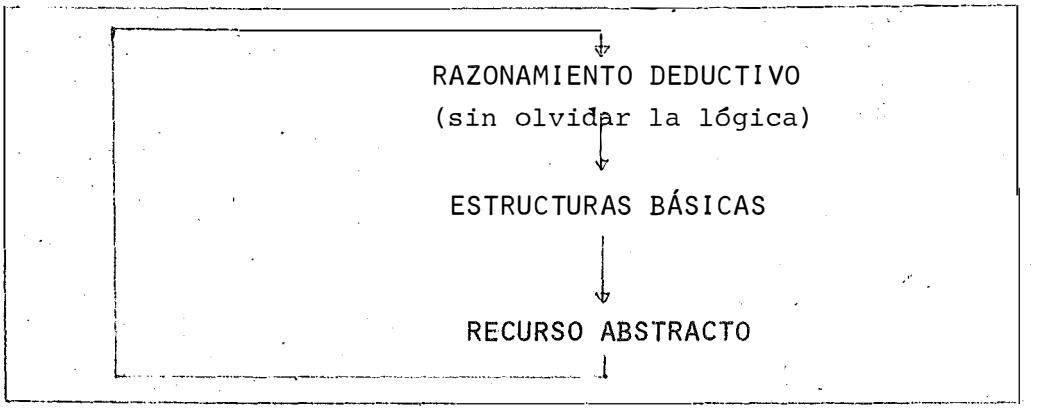

Lee datos iniciales: SEG., MIN, HORA, DIA, MES, N.
IMPRIME HORA Y FECHA
OSC = 0
OSC=OSC+1
SEG=59
NO
SI
SEG = 00
MIN=59
NO
SI
SEG=SEG+1
MIN = MIN+1
MIN = 00
HORA=23
NO
SI
HORA = HORA+1
HORA = 00
CALCULO "LIMITE MES"
DIA=LIM.
NO
SI
DIA=DIA+1
DIA = 01
MES=12
NO
SI
MES=MES+1
MES =01
IMPRIME HORA Y FECHA
OSC≠N
SI
NO
IMPRIME "SE ACABO LA BATERIA"

Fig. 5.

4. RESUMEN.

MÉTODO GENERAL DE DISEÑO DE PROGRAMAS ESTRUCTURADOS



CAPITULO 4

MAQUINAS DE TURING: DEFINICION, ESQUEMA FUNCIONAL Y EJEMPLOS.

0. INTRODUCCIÓN.

El propósito fundamental de este capítulo es definir - qué es, cómo funciona y cómo se diseña una máquina de Turing. A través de varios ejemplos se familiarizará el lector con la estructura de esta máquina, los alfabetos externo e interno, - su programación y las distintas representaciones de la misma - (lista de quíntuplas, esquema funcional, diagrama de estados) y la representación y manejo de las informaciones en la cinta de la máquina.

1. DEFINICIÓN DE MÁQUINA DE TURING.

En el año 1.936, antes del advenimiento de los ordenadores, el matemático inglés A.M. Turing inventó una máquina - computadora de una increible simplicidad en su estructura lógica. El concepto de algoritmo puede muy bien estudiarse en términos de esquemas funcionales de máquinas de Turing ya que, como veremos, éstas permiten disecar los algoritmos en las operaciones más elementales que cabe imaginar.

La máquina de Turing se ha convertido prácticamente en la *piedra angular de la moderna teoría de algoritmos*. Siendo - una máquina ideal, que cada uno define y construye con papel y lápiz, no se ve afectada por los avances tecnológicos. Es, pues,

un invariante de la informática y también un símbolo. La aso-- ciación de informática profesional más prestigiosa del mundo, la A.C.M. (Association for Computing Machinery) discierne to-- dos los años el premio Turing entre los científicos que más - han contribuido a generar avances significativos en el domi-- nio de la informática.

Véase en la figura 1 cómo se representaba esta máquina en el Scientific American en 1.963. (Tomada de (H.Wang, 1.974))

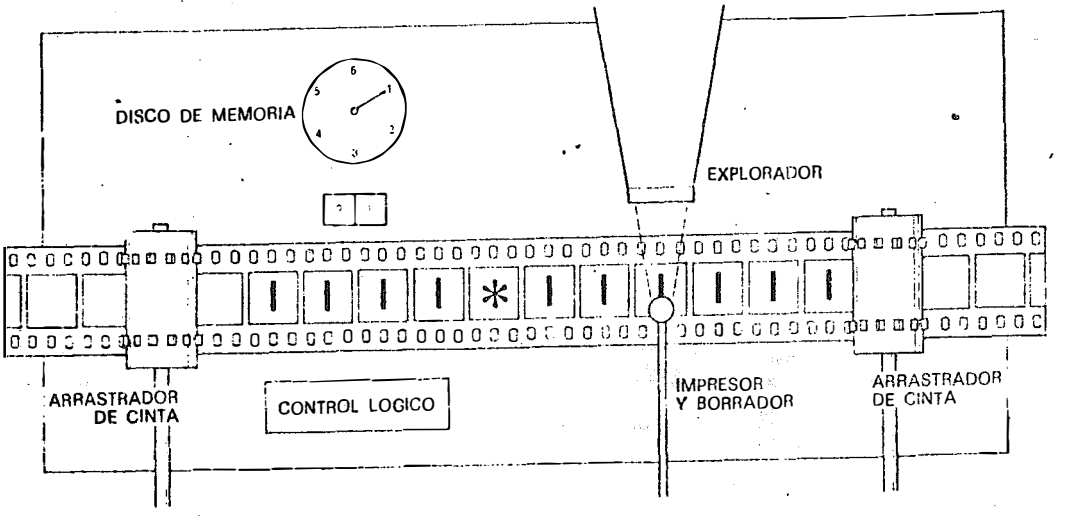


Fig. 1.

Una máquina de Turing es un autómata finito, junto con una cinta de longitud infinita (que en cualquier momento con-- tiene sólo un número finito de símbolos) dividida en casillas (cada casilla puede contener un solo símbolo o estar en blan-- co) y un aparato para explorar (leer) una casilla, imprimir un nuevo símbolo sobre ella, y mover la cinta una casilla a la de recha o a la izquierda. Veamos en detalle estos elementos:

a) *La cinta constituye una memoria infinita.* Los símbo

los que sobre ella están escritos o se pueden escribir e_i, - - $i \in I \subset N$ pertenecen a un conjunto finito E, al que llamamos - alfabeto externo de la máquina. Hagamos m = Card (E). Estos m símbolos sirven para codificar la información suministrada a - la máquina. El símbolo blanco, o vacío, ($\square$ generalmente, a - no ser que se designe expresamente por 0) forma parte del alfabeto, lo que no querrá decir que pueda considerarse infinita - la información contenida en la cinta. Por tanto, en un estadio cualquiera del funcionamiento de la máquina, toda información registrada sobre la cinta se presenta bajo la forma de una palabra escrita en el alfabeto externo E, a razón de un símbolo por casilla.

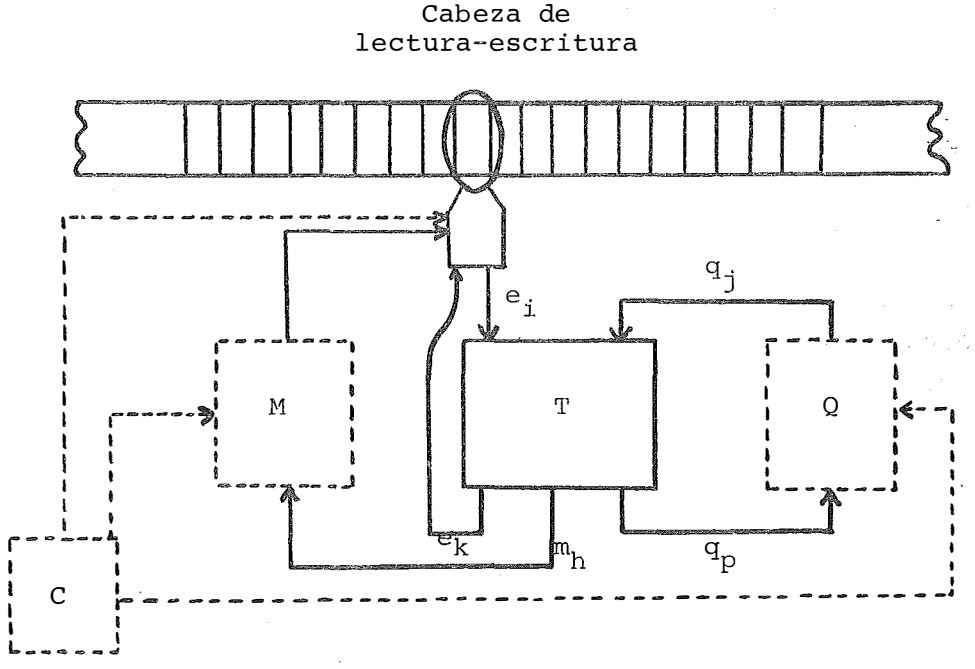


Fig. 2.

b) En principio, había una cabeza de lectura y escritura inmóvil y la cinta se desplazaba bajo la misma. Pero en este texto, -y no somos los únicos en hacerlo así-, supondremos de ahora en adelante que ocurre lo contrario: *la cinta estará*

inmóvil y se desplazará la cabeza, con lo que resultará más sencilla la representación gráfica de la dinámica del contenido de la cinta. Dado que la cabeza se desplaza una posición a la derecha o a la izquierda o ninguna posición, es evidente que las instrucciones que puede ejecutar esta máquina tienen que estar compuestas de las especificaciones de una operación de lectura/escritura y de un movimiento (derecha ($\rightarrow$), izquierda ($\leftarrow$), inmóvil ($\leftrightarrow$); conjunto M, M = {$\rightarrow$, $\leftarrow$, $\leftrightarrow$ }. (Observación: puede definirse la M.T. con M = {$\rightarrow$, $\leftarrow$} pero aquí se ha optado por la primera versión, más flexible).

c) El bloque T, que puede atravesar diferentes estados q_j, de un conjunto finito Q, está conectado a la cabeza de lectura/escritura por un enlace de entrada (lectura, símbolo e_i) y un enlace de salida (escritura, símbolo e_k). El estado le es presentado por el bloque Q. La función lógica realizada por el bloque T hace corresponder a la pareja (e_i, q_j) un vector de salida (e_k, m_h, q_p) con e_i, $e_k \in E$, $m_h \in M$, q_j, $q_p \in Q$. Naturalmente, *el autómata, en un sentido formal, está constituído por los bloques T y Q*. Los bloques Q y M actúan como dos memorias internas, que conservan respectivamente el estado q_p y la orden de movimiento m_h, producidos por el autómata durante su trabajo, hasta el comienzo del instante siguiente, comienzo desencadenado por un secuenciador C, que controla los pasos o fases de trabajo. (Nota: aunque se designen por las mismas letras M y Q, no confundir bloque con conjunto. El contexto dirá de qué cosa se trata).

En un eje de tiempos se representan las fases del funcionamiento de una M.T. (fig. 3).

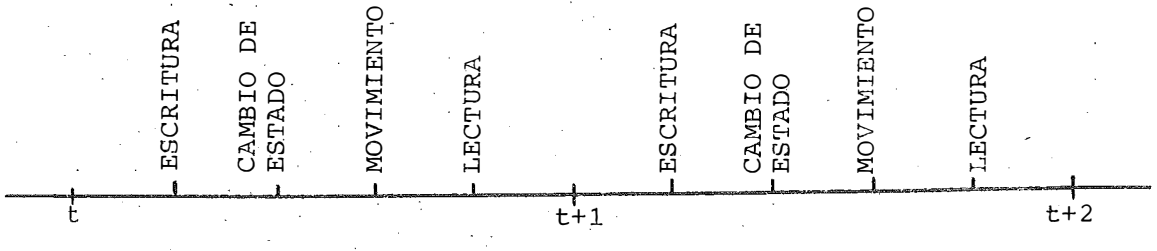


Fig. 3.

No es esencial el orden en que ocurran "cambio de estado" y "movimiento de la cabeza" que, incluso, pueden suponerse simultáneos.

Con lo dicho, podemos definir así el autómata finito - T - Q, que es el componente lógico de la Máquina de Turing:

$$T - Q = \langle E, (E \times M) \cup (Stop), Q, f, g \rangle \quad (1)$$

E conj. finito de símbolos en la cinta (alfabeto externo) (2)

M conj. finito de movimientos de la cabeza (3)

Q conj. finito de estados internos (alfabeto interno) (4)

$f: E \times Q \to Q$ función transición (5)

$g: E \times Q \to (E \times M) \cup (Stop)$ función de salida (6)

Cualquiera que sea la información, o palabra A escrita inicialmente en la cinta según el alfabeto E, pueden ocurrir - dos cosas:

1. Bien, al cabo de un número finito de pasos se detiene la máquina dando la señal de stop. Sobre la cinta se encontrará una cierta palabra B escrita según el mismo alfabeto, - que representa la información resultante. Se dice que la máquina es aplicable a la información A, y que transforma A en B.

2. Bien, la señal de stop no se produce nunca, en cuyo caso se dice que la máquina es inaplicable a la información - inicial A.

El funcionamiento de una particular y concreta máquina de Turing podrá describirse especificando los items (1) a (6), junto con los siguientes elementos:

información inicial registrada en la cinta (7)

posición inicial de la cabeza (8)

estado inicial del autómata (9)

o, lo que es lo mismo, mediante los items (7) a (9) y una tabla (también llamada esquema funcional) de todas las quíntuplas posibles $e_i\ q_j\ e_k\ m_h\ q_p$. Esta tabla es un programa para la máquina de Turing, cuyas instrucciones dicen: "si la máquina está en el estado q_j y lee el símbolo e_i, que escriba el símbolo e_k, mueva la cabeza una posición m_h y cambie al estado q_p".

(Nota: reflexione el lector que en la definición y esquema de la M.T. están presentes de una manera o de otra todos los subsistemas de la fig. 4 del capítulo 1).

2. FUNCIONAMIENTO DE LA MÁQUINA DE TURING A TRAVÉS DE LOS EJEMPLOS.

2.1. Suma de dos números enteros no nulos escritos en el alfabeto {|}.

La tabla adjunta expresa el programa correspondiente a un algoritmo de suma de dos números enteros no nulos para una máquina de Turing con las siguientes características:

$$E = \{\square, |, *\}$$

$$Q = \{q_0, q_1, q_2\}$$

información inicial en la cinta: los dos números enteros representados por grupos de | (ej. 3 ≡ |||; 5 ≡ |||||) separados por un asterisco *.

posición inicial de la cabeza: sobre el primer | de la izquierda

estado inicial del autómata: q_0

e_i	q_j	e_k	m_h	q_p
|	q_o	☐	→	q_2
|	q_1	|	←	q_1
|	q_2	|	→	q_2
☐	q_0	☐	→	q_0
☐	q_1	☐	→	q_0
☐	q_2	|	↔	q_1
*	q_0	☐	↔	stop
*	q_1	*	←	q_1
*	q_2	*	→	q_2

e_i \ q_j	q_o	q_1	q_2
|	☐ → q_2	| ← q_1	| → q_2
☐	☐ → q_0	☐ → q_0	| ↔ q_1
*	☐ ↔ stop	* ← q_1	* → q_2

Fig. 4. Dos representaciones tabulares alternativas

En el primer instante, la pareja de entrada es (|,q_0) lo que ocasiona un trío (☐ → q_2), esto es, se borra el primer palote, se desplaza la cabeza una posición a la derecha y se pasa al estado q_2. Y así una y otra vez. La mejor forma de apreciar la mecánica será con un caso numérico concreto, por ejem-

plo la suma de 2 y 3, situado el primero de ellos a la izquier da. Véanse en la figura 5 las configuraciones sucesivas de la suma 2 + 3 en una máquina de Turing.

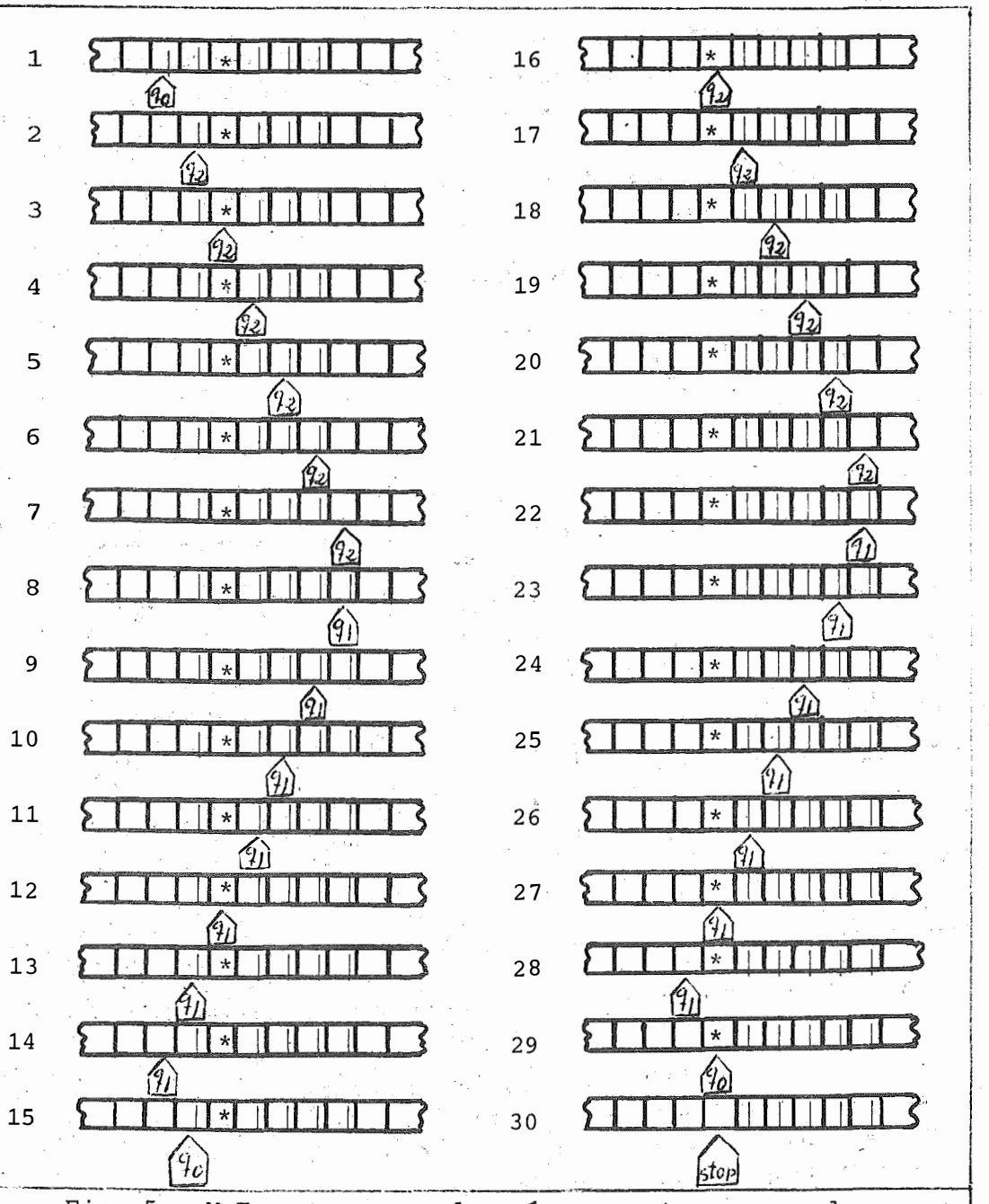


Fig. 5. M.T. para sumar dos números enteros no nulos.

La máquina se detiene en el movimiento nº 30. La información resultante, o solución del problema, o palabra B, aparece situada a la derecha de la cabeza cuando ésta se para.

Es evidente que la presentación de las configuraciones de la cinta bajo la forma de la figura 5 es poco manejable. Por tal razón se le prefiere las de la figura 5' y 5'', en las que prácticamente siempre el símbolo'blanco' sería el cero.

CINTA	ESTADO
00⊥|*|||00	q_0
000⊥*|||00	q_2
000|*⊥||00	q_2
000|*|⊥|00	q_2
000|*||⊥00	q_2
000|*|||⊥0	q_1
000|*||⊥|0	q_1
000|*|⊥||0	q_1
000|*⊥|||0	q_1
000|*⊥|||0	q_1
000⊥*|||0	q_1
000⊥*|||0	q_1
000⊥*|||0	q_0
0000*⊥|||0	q_2
0000*⊥|||0	q_2
0000*|⊥||0	q_2
0000*||⊥|0	q_2

CINTA	ESTADO
0000*|||1̲0	q_2
0000*||||0̲	q_2
0000*||||1̲	q_1
0000*|||1̲|	q_1
0000*||1̲||	q_1
0000*|1̲|||	q_1
0000*1̲||||	q_1
0000*̲|||||	q_1
0000̲0*|||||	q_1
0000*̲|||||	q_0
00000̲|||||	stop.

Fig. 5'

$00q_0||*|||00$
$000q_2|*|||00$
$000|q_2*|||00$
$000|*q_2|||00$
$000|*|q_2||00$
$000|*||q_2|00$
$000|*|||q_200$
$000|*|||q_1|0$
$000|*||q_1||0$
$000|*|q_1|||0$
$000|*q_1||||0$
$000|q_1*||||0$
$000q_1|*||||0$
$00q_10|*||||0$
$000q_0|*||||0$
$0000q_2*||||0$

Fig. 5''

A las expresiones de la figura 5'' se les llama descripciones instantáneas. La descripción instantánea da de manera absolutamente precisa el comportamiento de la M.T. En efecto, la descripción, si no se toma en cuenta q_j, da el contenido de la cinta y, puesto que el símbolo a la derecha de q_j en la expresión es e_i (por eso no es necesario el subrayado indicativo de la posición de la cabeza) se tiene además la pareja (e_i, q_j) representativa de la situación de trabajo en que se encuentra la máquina.

Aprovechemos este ejemplo y esta última representación para definir el cálculo de una Máquina de Turing (Gross, Lentin, 1.967).

Cálculo de una Máquina de Turing

Se dice que una descripción instantánea γ es terminal, relativamente a una máquina Z, si no existe ninguna descripción ω tal que $\gamma \rightarrow \omega$ ($\rightarrow$ significa aquí *es seguida por*)

Se llama cálculo de una M.T. determinada a una sucesión:

$$\gamma_1, \gamma_2, \ldots\ldots \gamma_p$$

de descripciones instantáneas tales que $\gamma_i \rightarrow \gamma_{i+1}$, para $1 \leq i < p$ con γ_p terminal. Se puede escribir

$$\gamma_p = \text{Res. Z } (\gamma_1), \qquad (10)$$

lo que se lee: "γ_p es el resultado de γ_1 para la máquina Z".

Nota: Obsérvese que la expresión (10) es una forma más completa de decir que la máquina Z es aplicable a una determinada información, ya que incluye el estado y la posición de la cabeza

en la situación inicial y en la final.

2.2. Algoritmo de Euclides para el cálculo del m.c.d. de dos números enteros escritos en el alfabeto {|}.

$E = \{\square, |, *, \alpha, \beta\}$

$Q = \{q_0, q_1, q_2, q_3, q_4\}$

información inicial en la cinta: los dos números enteros representados por grupos de |, separados por un asterisco *.

posición inicial de la cabeza: sobre el primer | del número de la izquierda.

estado inicial del autómata: q_0.

Esquema funcional (los recuadros vacíos significan combinaciones imposibles o indiferentes).

e_i \ q_j	q_0	q_1	q_2	q_3	q_4
|	$\beta \to q_0$	$\alpha \leftrightarrow q_2$	$\beta \leftrightarrow q_1$	$\| \to q_1$	$\| \leftarrow q_1$
$\square$	$\square \to q_3$	$\square \to q_4$	$\square \leftarrow q_3$	$\square \to q_1$	$\square \leftrightarrow$ stop
*	$\alpha \leftarrow q_0$		$\| \to q_2$	$\square \to q_2$	
α		$\alpha \leftarrow q_1$	$\alpha \to q_2$	$\| \leftarrow q_3$	$\square \to q_4$
β	$* \leftarrow q_0$	$\beta \leftarrow q_1$	$\beta \to q_2$	$\square \leftarrow q_3$	$\| \to q_4$

Fig. 6. M.T. para el algoritmo de Euclides aplicado a dos números enteros en {|}

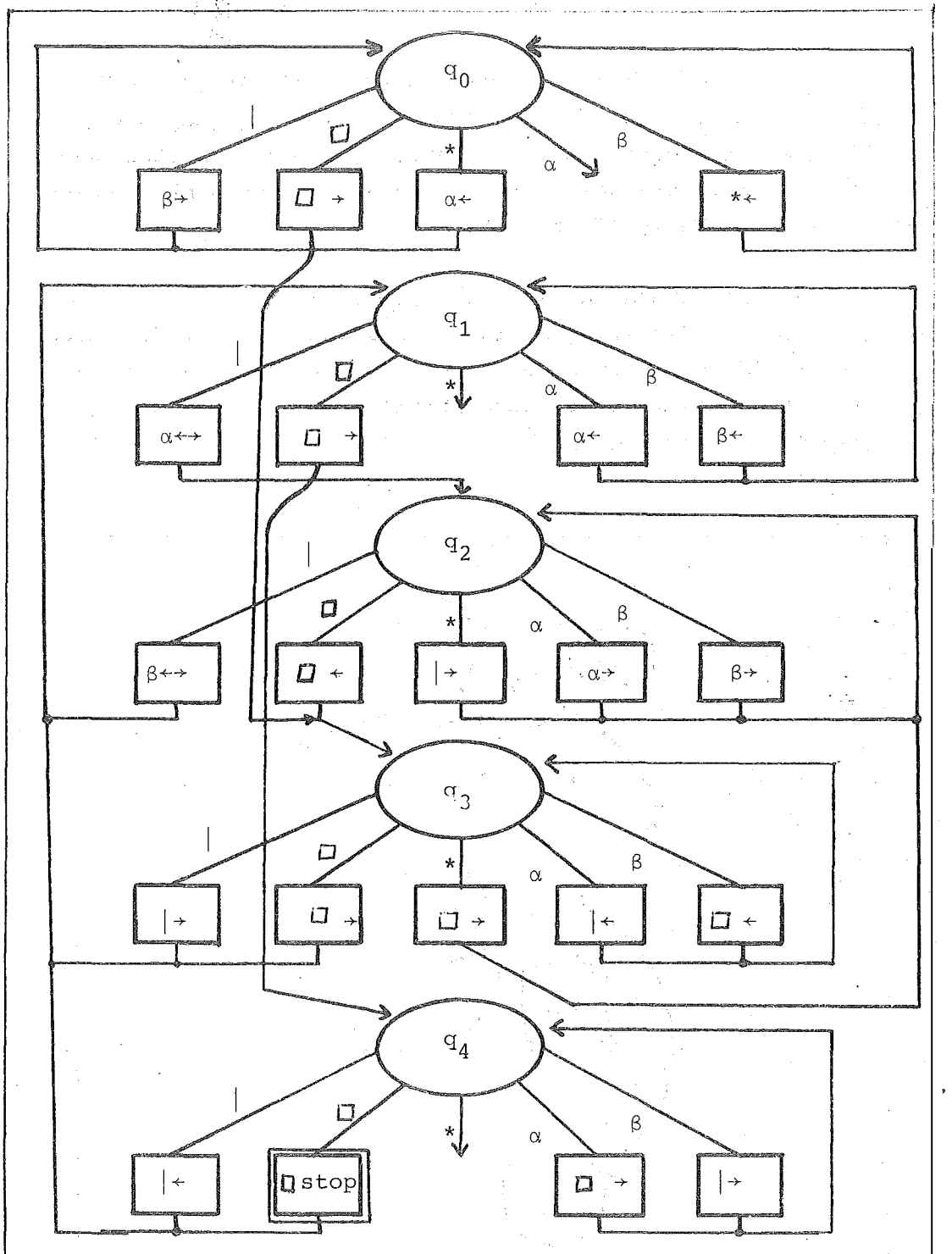


Fig. 7. Diagrama de estados de una Máquina de Turing para el cálculo del m.c.d. de dos números enteros. Los rectángulos representan la acción sobre el contenido y la posición relativa de la cinta.

La misma información que hay en el esquema funcional - puede expresarse mediante un diagrama de estados que no es, en definitiva, otra cosa que un organigrama de los que típicamente se usan en programación (fig. 7) o en la representación de una máquina de Mealy (figura 8).

Sigamos ahora la explicación que da Corge (Corge,1975) del proceso de cálculo, para lo que nos puede servir de ayuda la ilustración de algunos instantes significativos en el procesamiento de la cinta en el caso particular de los números 3 - (izquierda) y 6 (derecha).(Véase figura 9).

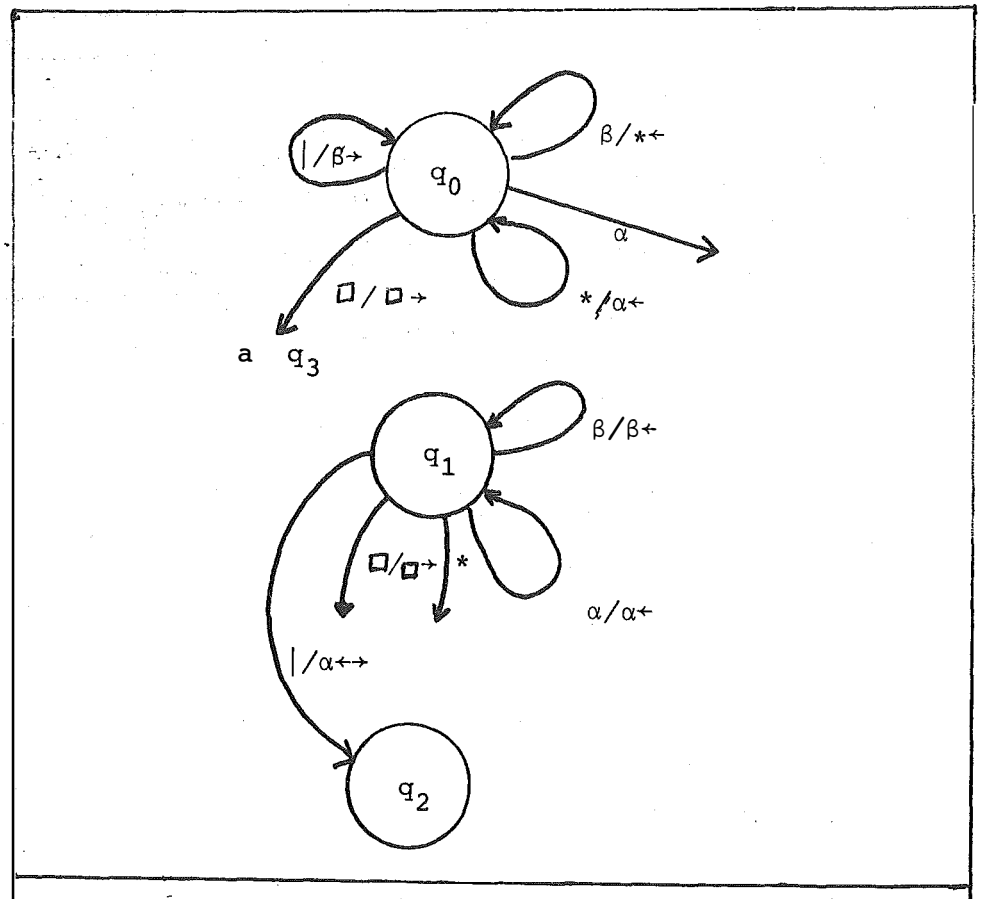
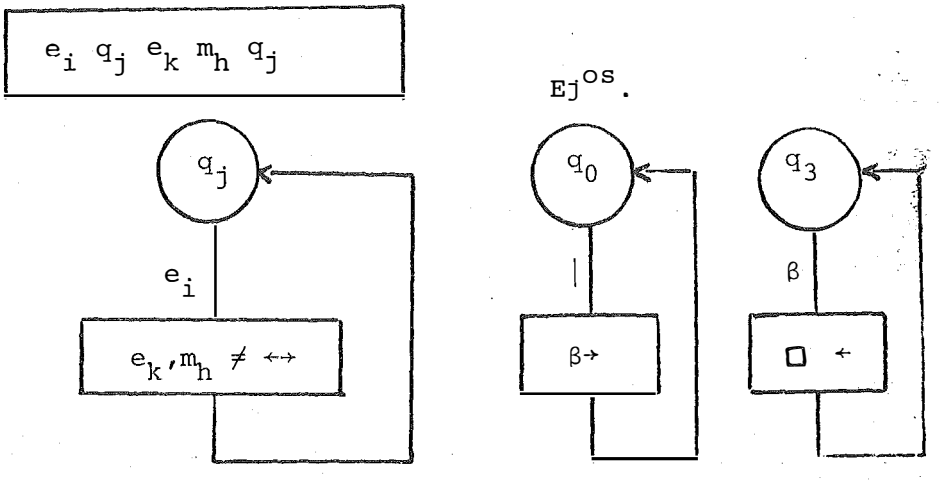


Fig. 8. Diagrama (incompleto) de estados en forma de máquina de Mealy.

La primera fase del proceso consiste en suprimir el asterisco que separa los dos números para hacer como si se quisiera yuxtaponer éstos, de manera que formen una palabra escrita en el alfabeto {|}, y en disponer la cinta de suerte que quedase bajo la cabeza el último símbolo |, que se habrá sustituido previamente por α. Esta fase acaba en el movimiento número 13 (¡atención, esto sólo es cierto en el caso práctico considerado!)

El diagrama de estados permite poner de manifiesto la existencia de dos tipos de bucles, que son aquellos procesos peculiares en que la máquina puede permanecer repetidamente en un mismo estado:

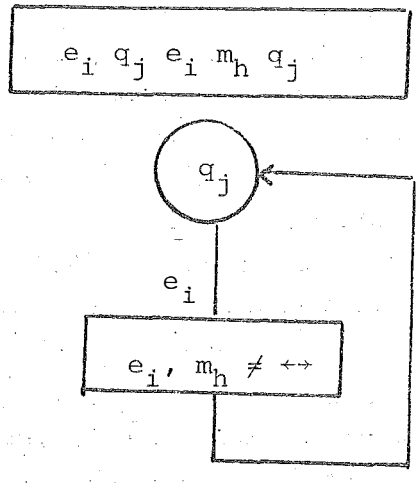
a)



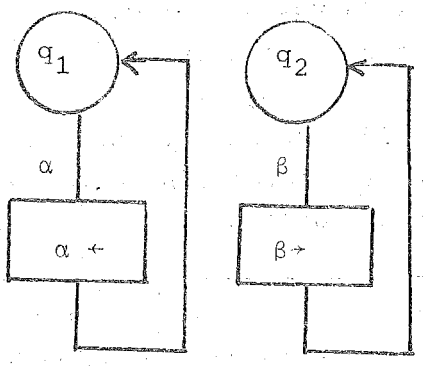
Ej^{os}.

Se sustituye un símbolo e_i por otro símbolo distint e_k sistemáticamente, mientras queden casillas relle con e_i en la dirección de exploración.

b)



Ej^{os}.



Se explora la cinta en una determinada dirección sin - producir cambios en la información que contiene, mientras la cabeza lea un símbolo determinado e_i.

A esta primera fase siguen los ciclos de comparación y sustracción que movilizan respectivamente los estados q_1, q_2 y los q_3, q_4.

Para comparar los dos números y detectar cual de ellos es mayor, la máquina examina los dos grupos de |, marcando con un símbolo diferente (α para el número de la izquierda, β para el de la derecha), alternativamente un símbolo | de cada grupo. El punto de partida del ciclo de comparación es el movimiento 13, en que ya se ha sustituido un primer símbolo | del número de la izquierda por α. Dos instantes más tarde (configuración 15), el primer símbolo | del segundo número ha sido sustituido por β. El proceso sigue así, en sustituciones alternadas, hasta la configuración 33, en la que vemos que ya no hay más sustituciones posibles en el primer número. Pero esto la máquina no lo descubre hasta transcurridos 6 instantes más (configuración 39).

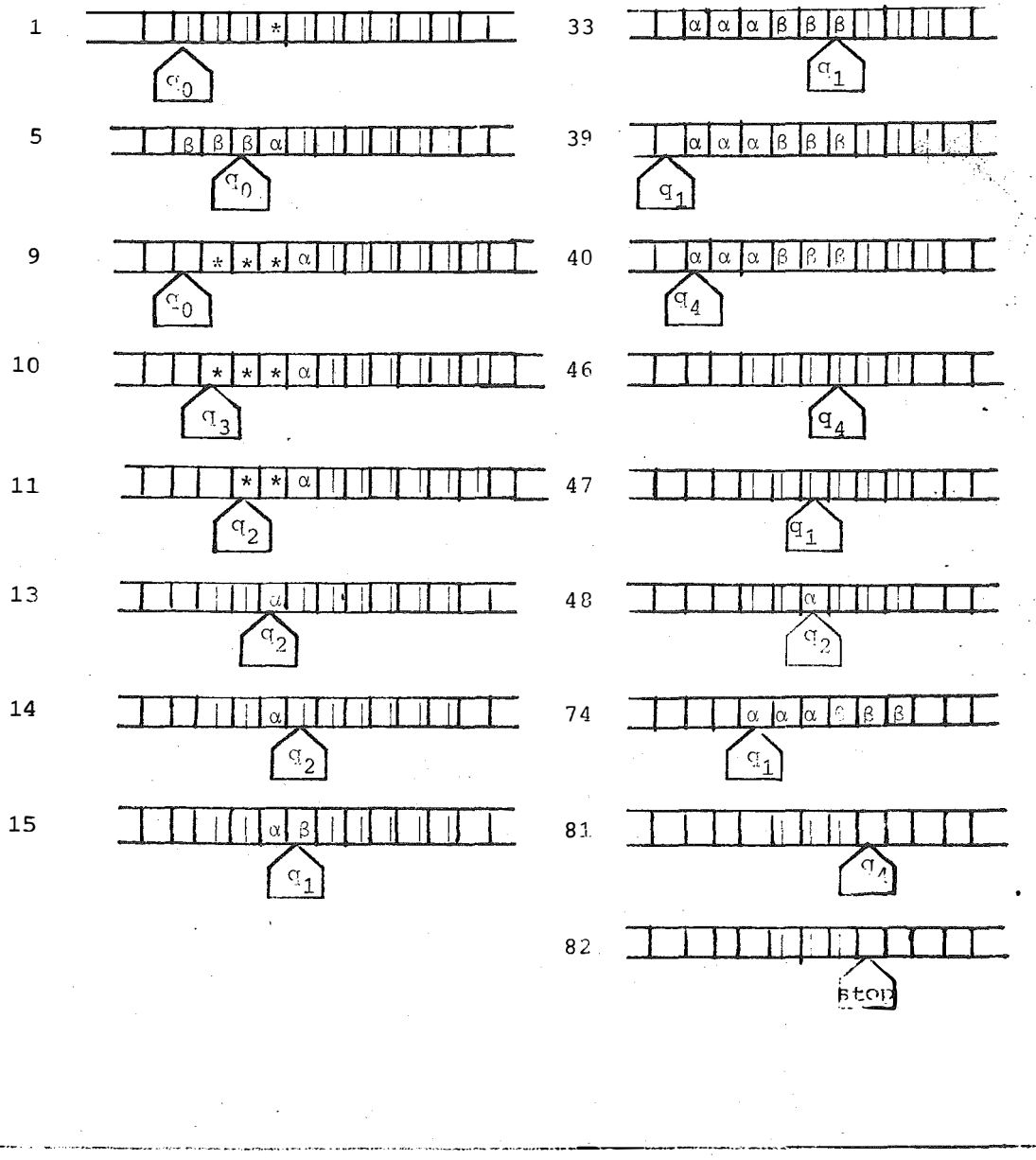


Fig. 9. Instantes significativos en el proceso de cálculo del m.c.d. de 3 y 6 por una máquina de Turing.

El estado q_4 provoca, cuando en la cinta hay α o β, sucesivos desplazamientos a derecha con sustitución de α por □ (borrado) y de β por 1, respectivamente. Así se llega a la configuración 46 y un instante después, a la 47. Desde la configuración 40 a la 47 se ha desarrollado un ciclo de sustracción que, si no ha hecho uso del estado q_3 se debe, una vez más, a las características del caso práctico escogido. (El lector puede ensayar con otro par de números. Así se identificará con el funcionamiento de una máquina de Turing y, de paso ¡y con paciencia!, podrá verificar si el programa dado para el cálculo del m.c.d. es o no es correcto).

La configuración 48 marca el principio de un nuevo ciclo de comparación hasta el momento 74. Seguidamente tiene lugar otro ciclo de sustracción, hasta el instante 81. Las alternancias de ciclos de comparación y de sustracción seguirían desarrollándose hasta llegar a tener dos números iguales, en cuyo momento se acaba el proceso. Esto es precisamente lo que ya ha ocurrido en el caso del 3 y del 6, y en la configuración 82 se detiene la máquina, quedando el resultado escrito a la izquierda de la cabeza: 3.

2.3. Cálculo del m.c.d. de dos números enteros escritos en D (D = {0,1,2,3,....9}) por el procedimiento general de construir un programa a base de subprogramas. Composición de máquinas de Turing.

Ahora nos podemos proponer definir una M.T., y su correspondiente programa, para calcular el m.c.d. de dos números enteros expresados en D = {0,1,2,3,....8,9} dando el resultado en el mismo alfabeto.

Esto se puede hacer por el sistema de elaborar un diseño original, considerando el problema en su totalidad, lo que

no resulta demasiado fácil. También puede descomponerse el problema en partes más sencillas, resolver éstas y después ensamblar las soluciones en un conjunto coherente. Esta técnica es la de composición de M.T.'S. Vamos a seguir este segundo procedimiento, que es muy común en informática, debido al menos a estas tres razones:

a) Es más sencillo.

b) Es más fiable.

c) Es más económico, por ser más sencillo, por ser más fiable y porque puede aprovecharse trabajo ya desarrollado anteriormente.

Y aquí, adicionalmente, es más educativo. Se utilizará como subprograma básico la M.T. definida en el apartado 2.2.

La información inicial de la cinta será $N_2 * N_1$, con N_1, $N_2 \in D$. Si se pretende aprovechar el diseño de M.T. de 2.2 será necesario crear otros subprogramas (de manera más rigurosa, otras M.T.) con la secuencia general expresada por el organigrama de la figura 10.

(No se olvide que todas las máquinas de Turing son estructuralmente idénticas, lo que se materializa por el formato constante y el significado de sus instrucciones $e_i\ q_j\ e_k\ m_h\ q_p$. En virtud de ello es posible aspirar a fundir en una sola M.T. varias M.T. distintas, siempre que se compatibilicen sus elementos diferenciales o condiciones de contorno: los alfabetos, las condiciones iniciales, las funciones f,g y las condiciones terminales).

Adaptamos y corregimos de Corge (Corge, 1.975) los esquemas de M.T.0, M.T.1 y M.T.3. Para aligerar los contenidos de los esquemas vamos a convenir en suprimir los símbolos de -

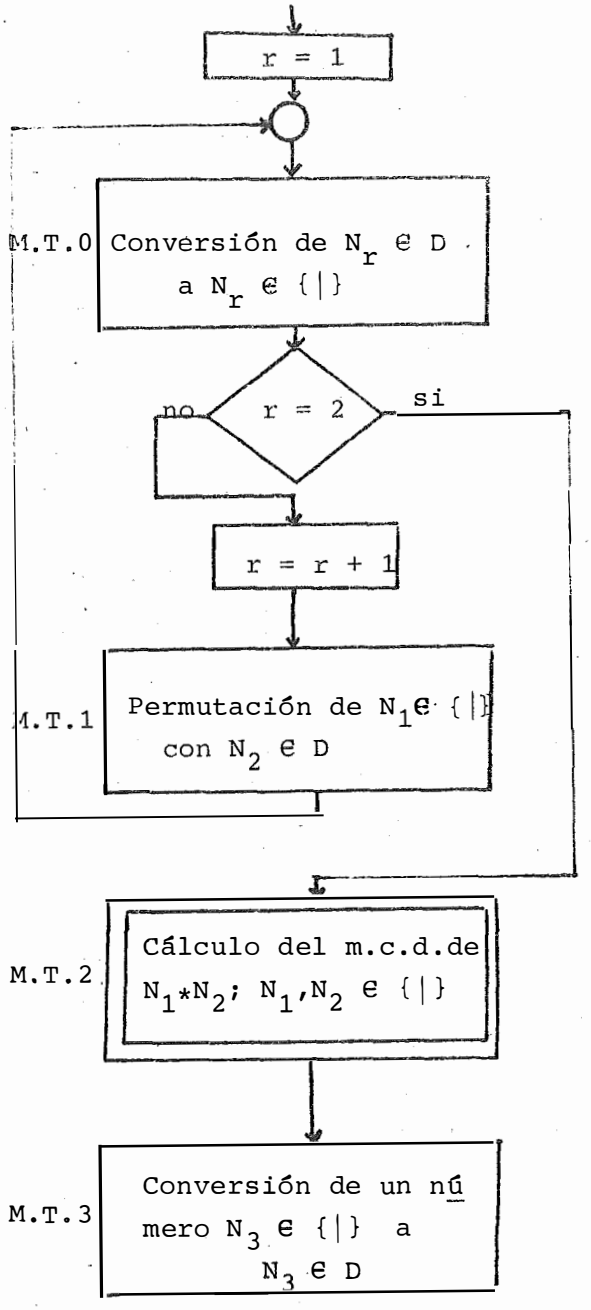

r = 1
M.T.0
Conversión de $N_r \in D$ a $N_r \in \{|\}$
no
r = 2
si
r = r + 1
M.T.1
Permutación de $N_1 \in \{|\}$ con $N_2 \in D$
M.T.2
Cálculo del m.c.d.de $N_1 * N_2$; $N_1, N_2 \in \{|\}$
M.T.3
Conversión de un número $N_3 \in \{|\}$ a $N_3 \in D$

Fig. 10

escritura y de estado siguiente cuando sean iguales a los de - entrada y el de desplazamiento cuando sea $\leftrightarrow$. (Reproducimos - M.T.2, también con este convenio. Una casilla vacía indica una combinación imposible, ya que nunca se puede producir ésta por causa del convenio).

e_i \ q_j	q_0	q_1	q_2
0	9 $\leftarrow$	$\rightarrow$	
1	0 $\rightarrow q_1$	$\rightarrow$	
2	1 $\rightarrow q_1$	$\rightarrow$	
3	2 $\rightarrow q_1$	$\rightarrow$	
4	3 $\rightarrow q_1$	$\rightarrow$	
5	4 $\rightarrow q_1$	$\rightarrow$	
6	5 $\rightarrow q_1$	$\rightarrow$	
7	6 $\rightarrow q_1$	$\rightarrow$	
8	7 $\rightarrow q_1$	$\rightarrow$	
9	8 $\rightarrow q_1$	$\rightarrow$	$\square$ $\rightarrow$
|	$\leftarrow$	$\rightarrow$	stop
$\square$	$\rightarrow q_2$	| $\leftarrow q_0$	$\rightarrow$

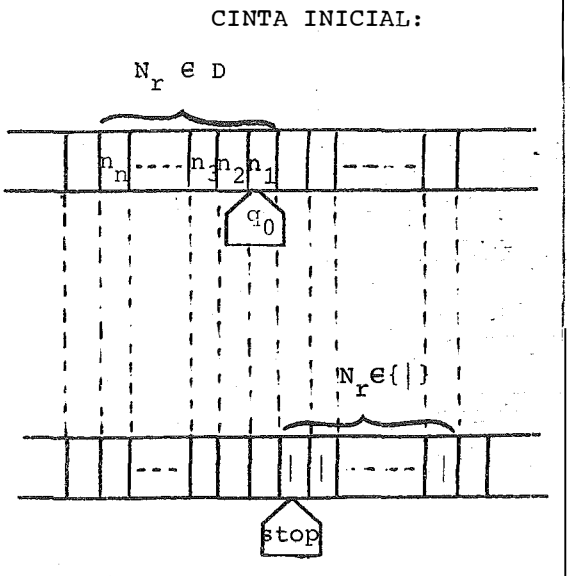


Fig. 11. M.T.0. CONVERSION DE $N_r \in D$ A $N_r \in \{|\}$

e_i \ q_j	q_0	q_1	q_2	q_3
0	→	→	←	←
1	→	→	←	←
2	→	→	←	←
3	→	→	←	←
4	→	→	←	←
5	→	→	←	←
6	→	→	←	←
7	→	→	←	←
8	→	→	←	←
9	→	→	←	←
|		→	←	□←q_2
□		← q_3	|→ q_1	* →q_1
*	← q_3	→	←	□←stop

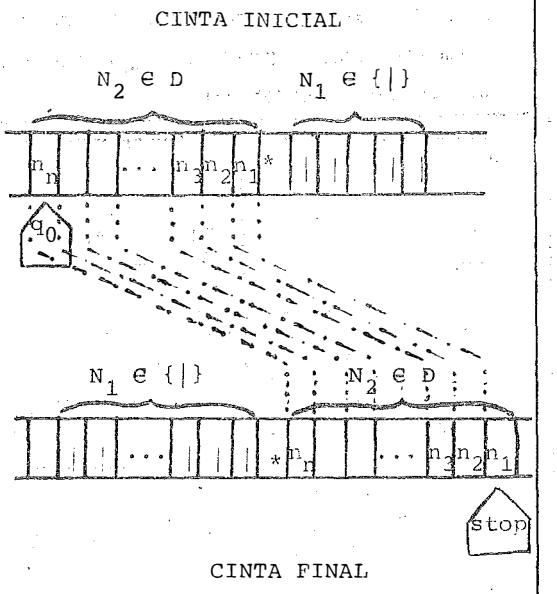


Fig. 12. M.T.1. PERMUTACION DE $N_1 \in \{|\}$ CON $N_2 \in D$

e_i \ q_j	q_0	q_1	q_2	q_3	q_4
|	β→	αq_2	βq_1	→q_1	←q_1
□	→q_3	→q_4	←q_3	→q_1	stop
*	α←		|→	□ → q_2	
α		←	→	|←	□ →
β	*←	←	→	□←	|→

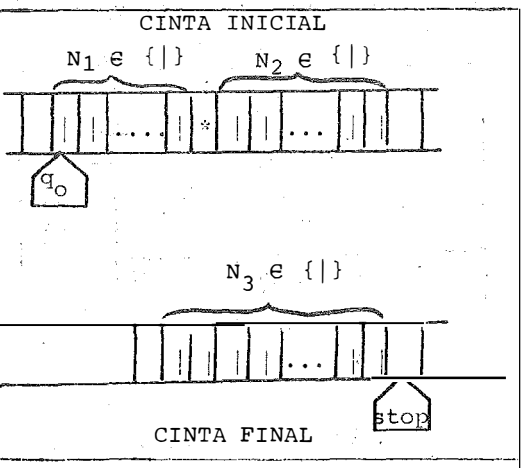


Fig.13. M.T.2. CALCULO DEL M.C.D. DE N_1* N_2;N_1,$N_2 \in \{|\}$

Se recomienda al lector tome los esquemas M.T.0, M.T.1 y M.T.3, que resuelven tres problemas distintos, como ejercicios independientes de análisis del funcionamiento de máquinas de Turing y, al menos, realice el esfuerzo de captar la idea global del proceso de composición de varias M.T., según se expone a continuación. El esquema de la figura 16 puede considerarse como letra pequeña, aunque sería de aconsejar su estudio detallado para aquellos lectores que gusten de profundizar algo más.

e_i \ q_j	q_0	q_1	q_2
0	1 q_2	stop	→
1	2 q_2	stop	→
2	3 q_2	stop	→
3	4 q_2	stop	→
4	5 q_2	stop	→
5	6 q_2	stop	→
6	7 q_2	stop	→
7	8 q_2	stop	→
8	9 q_2	stop	→
9	0 ←	stop	→
|	←	□ ← q_0	→
□	1 q_2	stop	← q_1

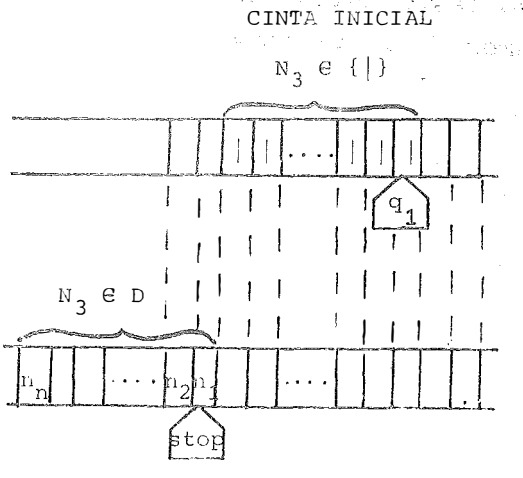


Fig. 14. M.T.3. CONVERSION DE $N_3 \in \{|\}$ a $N_3 \in D$

Para encajar las cuatro máquinas en una sola y definitiva que resuelva el problema que nos hemos planteado, hay que introducir los cambios oportunos -amén de rebautizar los nombres de los estados al objeto de referirlos a un alfabeto interno más amplio- tendentes a adaptar las diferencias mencionadas entre la forma cómo empieza un subprograma y la forma cómo empieza el siguiente (incluyendo las diferencias de alfabeto externo entre máquina y máquina). La figura 15 recoge la configuración de la cinta de datos antes y después de aplicarle la M.T. correspondiente al proceso indicado en el organigrama de la figura 10, con señalamiento expreso de las modificaciones que hay que producir en aquella para ajustar la situación final de una máquina con la inicial de la siguiente.

En la figura 16 damos el esquema funcional de la M.T. solución (que esperamos sea correcto), utilizando superíndices en los nombres de los estados para que el lector vea de qué subprograma procede cada uno de ellos y recuadrando con trazo grueso las combinaciones introducidas para la adaptación a que acabamos de referirnos.

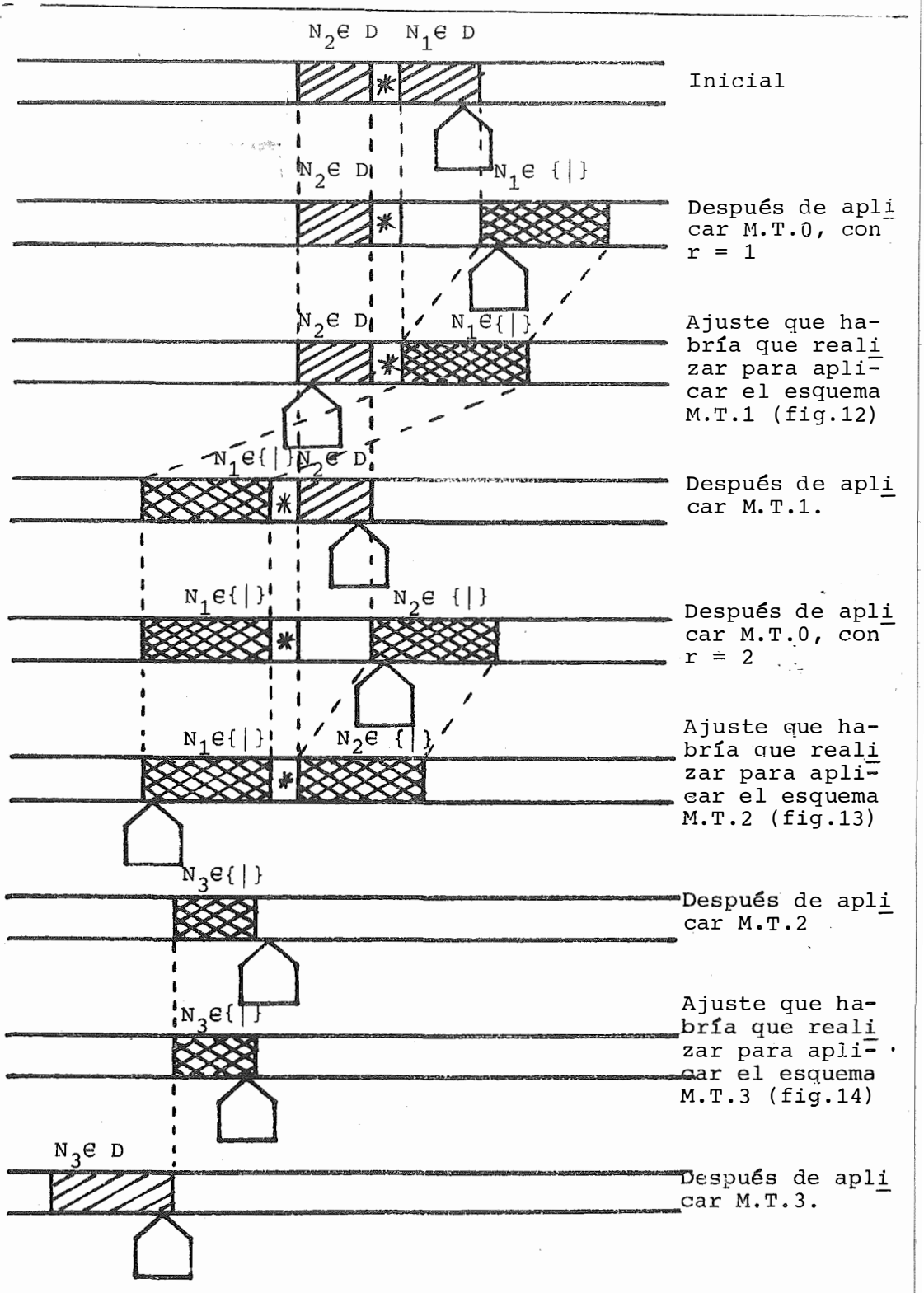


Fig. 15. SUCESIVAS CONFIGURACIONES DE LA CINTA AL APLICAR EL PROCESO DE LA FIGURA 10.

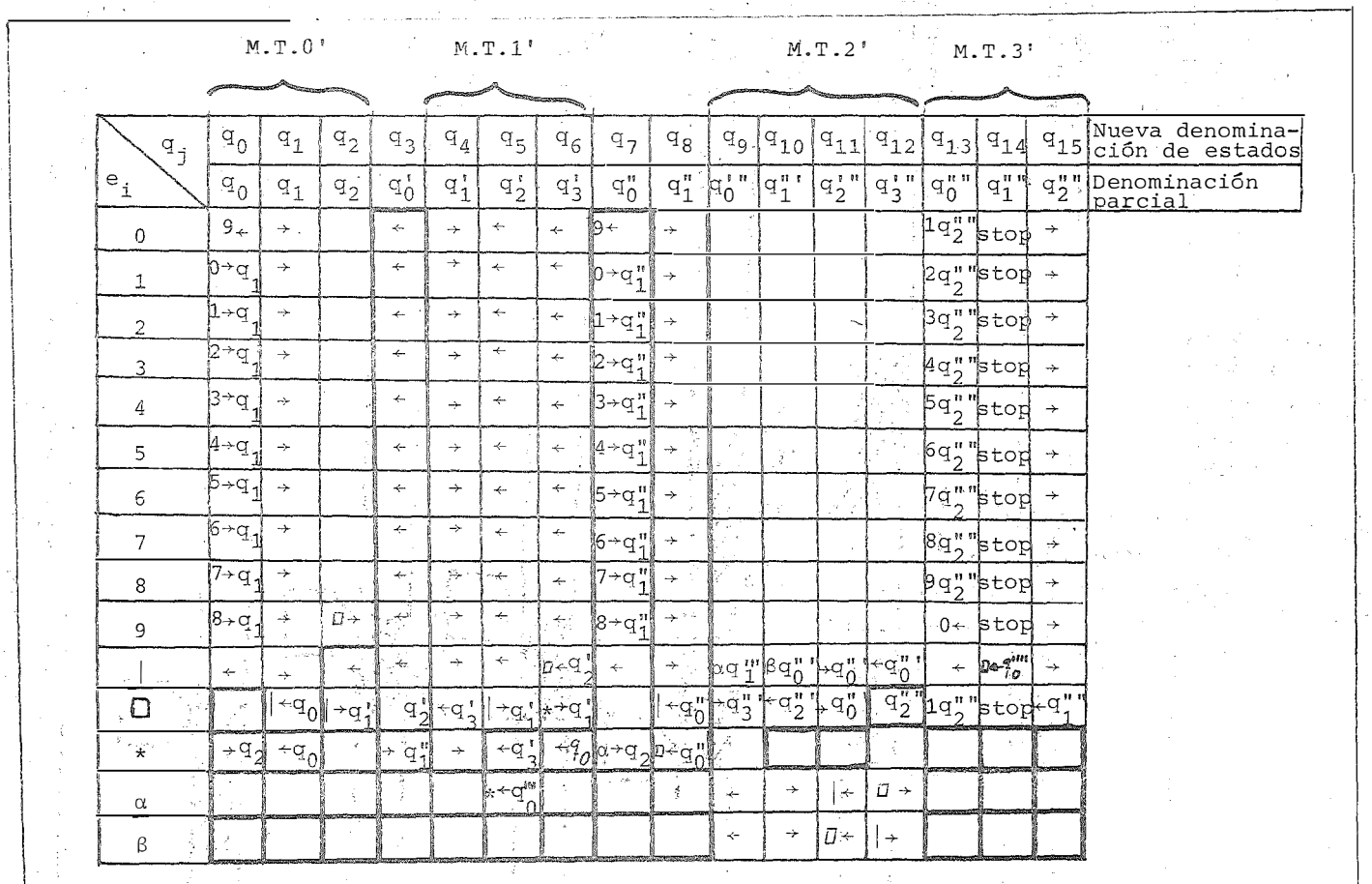


Fig. 16. ESQUEMA FUNCIONAL DE UNA M.T. PARA EL CALCULO DEL M.C.D. DE N_1 Y N_2; INFORMACION INICIAL $N_2 * N_1$, N_1 Y $N_2 \in D$.

3. Diseño de una máquina de Turing.

En el apartado anterior hemos estudiado, partiendo de unos datos en la cinta, el funcionamiento de la máquina ya diseñada. *Otro problema mucho más complejo es el de especificar precisamente las características de una M.T. para resolver un tipo de problema, esto es, definir los conjuntos Q, E, las funciones f,g, y las configuraciones iniciales y finales de la cinta.* Es el problema del diseño, para el cual no existe otra metodología que el propio razonamiento lógico.

En este apartado se verá un nuevo ejemplo de M.T. resaltándose la estrategia seguida en el razonamiento para su creación. Se trata de diseñar una M.T. con cinta direccionable por etiqueta (Scala, Minguet, 1.974). (Después de estudiar este apartado, se le recomienda al lector reexamine las máquinas de Turing de las páginas anteriores a través de la óptica de diseño).

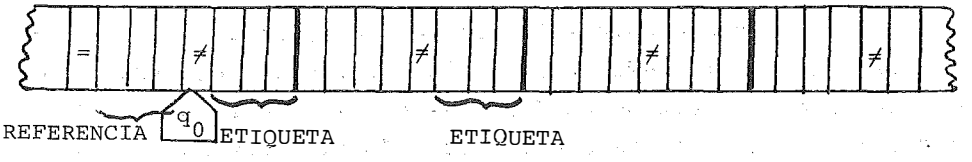


Fig. 17.

Suponemos que la cinta es binaria (esto no resta generalidad al problema) y que sobre ella se tienen unos registros de longitud fija, separados por el símbolo ≠. Estos registros están constituidos por una etiqueta, igualmente de longitud fija, y la información, sin que entre ambas exista ninguna separación.

A la izquierda del primer registro, entre el símbolo ≠ de éste y otro =, se suponen escritos los bits correspondientes a la etiqueta del registro que se quiere localizar, conjunto de bits al que llamaremos "referencia". La cabeza de lectura se sitúa inicialmente sobre el primer ≠, con la máquina en un estado q_0 (fig. 17).

El trabajo de la máquina consistirá en acudir a la referencia, recordar un bit y sustituirlo por otro símbolo (0 por α, 1 por β). Memorizado este bit, se desplazará la cabeza hacia la derecha en busca del primer registro no explorado y, dentro de él, del bit homólogo. Si coinciden bit de referencia y bit homólogo, éste último deberá ser anulado por la cabeza, que escribirá en su lugar α o β, volviendo a buscar el siguiente bit de referencia.

En el momento en que se encuentre una discrepancia, la máquina anulará todo el registro, volverá a =, restaurará todos los bits numéricos de la referencia en una pasada que le dejará sobre ≠ en condiciones de iniciar el proceso.

Si no se presentase la discrepancia, la cabeza no encontrará bits en la referencia, señal de que el registro ha sido localizado. La situación final es con la cabeza sobre el primer ≠ y toda la cinta escrita con símbolos literales (α, β) desde la referencia hasta la etiqueta del registro buscado (ambas inclusive). La primera información con bits es la deseada.

Veamos cómo se van asignando estados y construyendo el esquema funcional.

Con el estado q_0 se controla el retorno en el proceso de comparación de la referencia con una etiqueta, mediante un bucle a izquierdas, sin modificación de la información leida, hasta encontrarse con =, que conserva, iniciando un movimiento

a derechas (vease 1ª. columna del esquema).

El estado q_1 permite buscar el primer bit no anulado - de la referencia. Al encontrarlo, se memoriza, pasando a los - estados q_2, q_3, según sea 0 ó 1 y escribiendo α o β, espectiva mente.

Los estados q_2 y q_3, de movimiento a la derecha, deben ser insensibles a los bits que queden de la etiqueta controlan do sendos bucles que terminan cuando la cabeza lee el símbolo ≠, pasando la máquina al estado q_4 o q_5, según el caso.

Precisamente con los estados q_4 y q_5 habrá de buscarse, en movimiento a derechas, el primer bit no anulado del registro, cambiarlo por un símbolo α o β, y, si coincidiera con el bit -

e_i \ q_j	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8
0	←	α→q_2	→	→	α←q_0	α→q_6	α→	←	→
1	←	β→q_3	→	→	β→q_6	β←q_0	β→	←	→
α	←	→			→	→		←	0→
β	←	→			→	→		←	1→
=	→q_1							→q_8	
≠	←	stop	→q_4	→q_5	→	→	←q_7	←	←q_0

Fig. 18. MAQUINA DE TURING CON CINTA DIRECCIONABLE

memorizado de la referencia, retornar (mediante el estado q_0) para buscar otro nuevo bit no anulado de la referencia (estado q_1). Si no hubiera coincidencia, la máquina debe concluir que la comparación con este registro ha dado resultado negativo, - por lo que procederá a anular el resto del registro.

La anulación del resto del registro ha de controlarse por medio de un nuevo estado, q_6, con desplazamiento a dere- - chas hasta llegar al término del registro, expresado por el - símbolo $\neq$, en donde la M.T. cambiará de estado a q_7.

Se hace necesario un nuevo estado que llamamos q_7 para un retorno distinto del controlado por q_0, puesto que, habiendo resultado fallida una comparación referencia/etiqueta, es - preciso restaurar los bits anulados de la referencia para iniciar un nuevo proceso de comparación de aquella con la siguiente etiqueta en la cinta. Tal restauración se hará gracias a un estado q_8 al que se pasa, junto con un desplazamiento a la derecha, cuando la cabeza, en su desplazamiento de retorno a izquierdas (bucle sin escritura) controlado por q_7, lee el símbolo =.

(Observación: Nótese que se ha diseñado el esquema de manera que no se toma en cuenta el símbolo □ ,que sólo afectaría aquí a los bordes).

Como es natural, podría haberse seguido la técnica del organigrama para diseñar este proceso de cálculo. A fin de - - cuentas, el razonamiento lógico-literario que acaba de hacerse, el esquema funcional de la figura 18 y el organigrama de la figura 20 son equivalentes, si bien la segunda representación es, en el caso de la M.T., la más precisa y económica. Los organigramas de las figs. 19 y 20 son etapas sucesivas de un mismo razonamiento, el último más detallado y con especificación de estados.

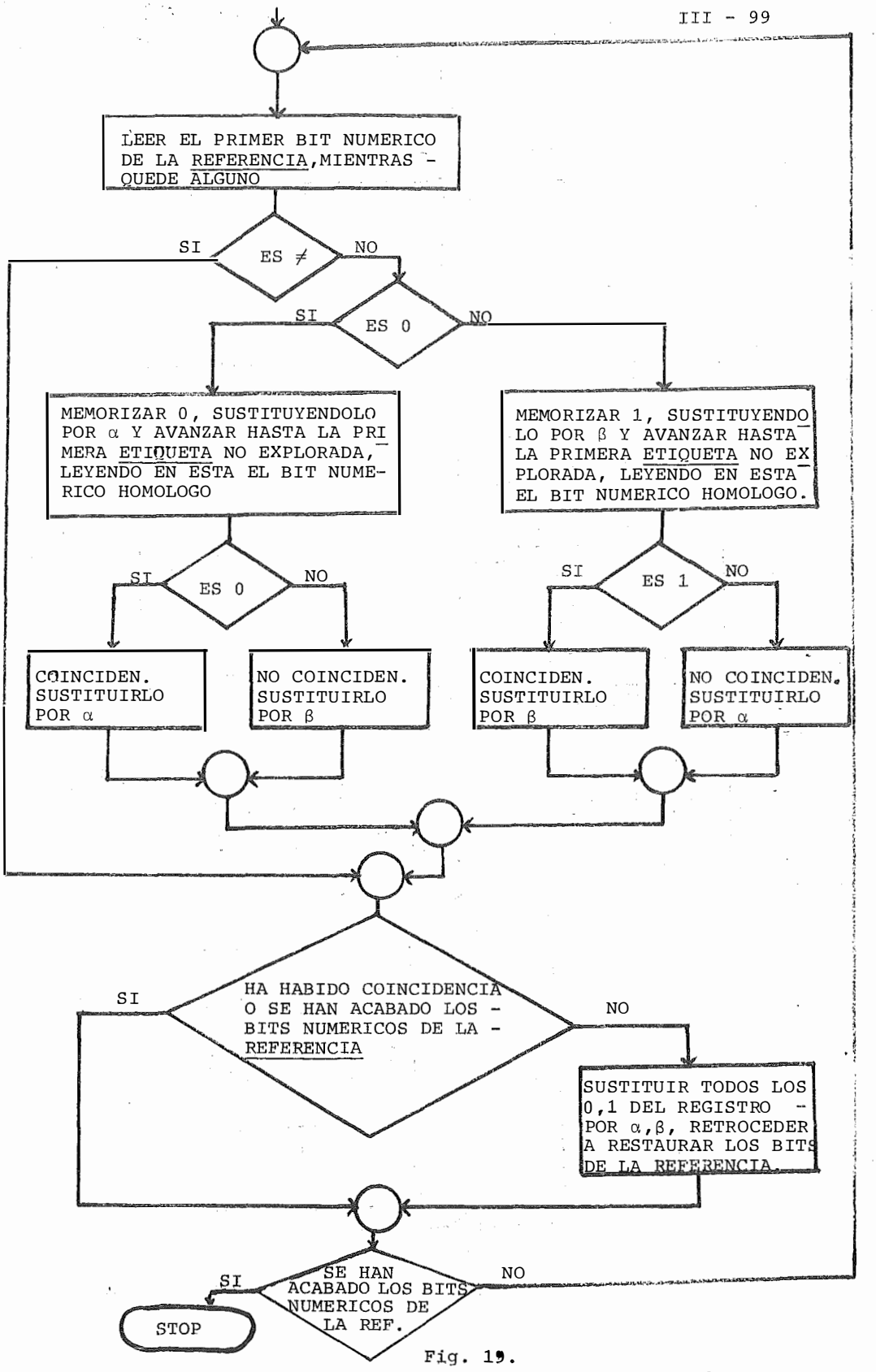

LEER EL PRIMER BIT NUMERICO DE LA REFERENCIA, MIENTRAS - QUEDE ALGUNO
SI
ES ≠
NO
SI
ES 0
NO
MEMORIZAR 0, SUSTITUYENDOLO POR α Y AVANZAR HASTA LA PRIMERA ETIQUETA NO EXPLORADA, LEYENDO EN ESTA EL BIT NUMERICO HOMOLOGO
MEMORIZAR 1, SUSTITUYENDOLO POR β Y AVANZAR HASTA LA PRIMERA ETIQUETA NO EXPLORADA, LEYENDO EN ESTA EL BIT NUMERICO HOMOLOGO.
SI
ES 0
NO
SI
ES 1
NO
COINCIDEN. SUSTITUIRLO POR α
NO COINCIDEN. SUSTITUIRLO POR β
COINCIDEN. SUSTITUIRLO POR β
NO COINCIDEN. SUSTITUIRLO POR α
SI
HA HABIDO COINCIDENCIA O SE HAN ACABADO LOS - BITS NUMERICOS DE LA - REFERENCIA
NO
SUSTITUIR TODOS LOS 0,1 DEL REGISTRO - POR α,β, RETROCEDER A RESTAURAR LOS BITS DE LA REFERENCIA.
SI
SE HAN ACABADO LOS BITS NUMERICOS DE LA REF.
NO
STOP

Fig. 19.

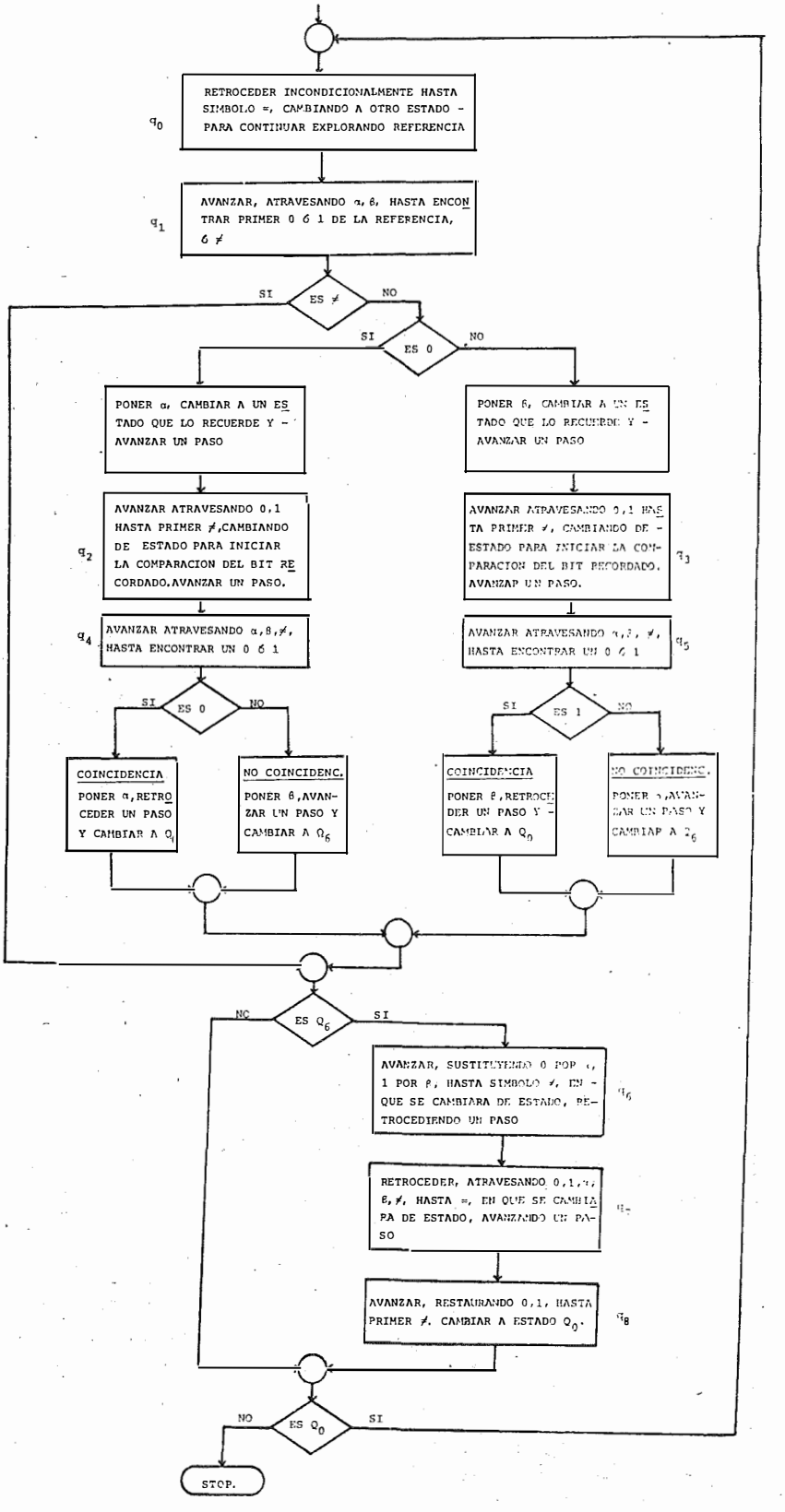

RETROCEDER INCONDICIONALMENTE HASTA SIMBOLO =, CAMBIANDO A OTRO ESTADO - PARA CONTINUAR EXPLORANDO REFERENCIA
q_0
AVANZAR, ATRAVESANDO α, β, HASTA ENCONTRAR PRIMER 0 Ó 1 DE LA REFERENCIA, Ó ≠
q_1
ES ≠
SI
NO
ES 0
SI
NO
PONER α, CAMBIAR A UN ESTADO QUE LO RECUERDE Y - AVANZAR UN PASO
PONER β, CAMBIAR A UN ESTADO QUE LO RECUERDE Y - AVANZAR UN PASO
AVANZAR ATRAVESANDO 0,1 HASTA PRIMER ≠,CAMBIANDO DE ESTADO PARA INICIAR LA COMPARACION DEL BIT RECORDADO.AVANZAR UN PASO.
q_2
AVANZAR ATRAVESANDO 0,1 HASTA PRIMER ≠, CAMBIANDO DE - ESTADO PARA INICIAR LA COMPARACION DEL BIT RECORDADO. AVANZAR UN PASO.
q_3
AVANZAR ATRAVESANDO α,β,≠, HASTA ENCONTRAR UN 0 Ó 1
q_4
AVANZAR ATRAVESANDO α,β,≠, HASTA ENCONTRAR UN 0 Ó 1
q_5
ES 0
SI
NO
ES 1
SI
NO
COINCIDENCIA
PONER α,RETROCEDER UN PASO Y CAMBIAR A Q_1
NO COINCIDENC.
PONER β,AVANZAR UN PASO Y CAMBIAR A Q_6
COINCIDENCIA
PONER β,RETROCEDER UN PASO Y - CAMBIAR A Q_0
NO COINCIDENC.
PONER α,AVANZAR UN PASO Y CAMBIAR A Q_6
ES Q_6
NO
SI
AVANZAR, SUSTITUYENDO 0 POR α, 1 POR β, HASTA SIMBOLO ≠, EN - QUE SE CAMBIARA DE ESTADO, RETROCEDIENDO UN PASO
q_6
RETROCEDER, ATRAVESANDO 0,1,α, β,≠, HASTA =, EN QUE SE CAMBIARA DE ESTADO, AVANZANDO UN PASO
q_7
AVANZAR, RESTAURANDO 0,1, HASTA PRIMER ≠. CAMBIAR A ESTADO Q_0.
q_8
ES Q_0
NO
SI
STOP.

Fig. 20.

Se le sugiere al lector la resolución del ejercicio siguiente: introducir en el esquema de la figura 18 las modificaciones necesarias y mínimas (sin ampliar el número de símbolos) para que la máquina siga realizando la misma función, pero deteniéndose en el símbolo ≠ que precede a la primera etiqueta - que coincida con la referencia. Nota: No deje de observarse - que la modificación pedida es del tipo de los ajustes que se - han requerido en la composición de máquinas del apartado 2.3.

4. Simulación de Máquinas de Turing, Máquina de Turing universal y otras consideraciones.

4.1. Simulación de M.T. por ordenador.

Diseñar una M.T. para resolver un problema medianamente complejo es asunto difícil, pero desarrollar su funciona- - miento a base de papel y lápiz es algo absolutamente tedioso. Lo primero podría tomarse como un desafío intelectual, como - una prueba (necesaria y no suficiente) demostrativa del grado de capacidad lógica de un futuro especialista en informática, como un juego de sociedad en grupos humanos altamente intelectualizados o como entretenimiento inagotable de náufragos, solitarios y presos políticos. Lo segundo es un trabajo rutina-- rio con una mecánica que, una vez conocida, no aporta nada especial al individuo e incluso puede resultar psicológicamente casi inaceptable si se dispone de ordenadores o cuando menos, de las pequeñas y potentes calculadoras electrónicas de bolsillo.

Así pues, simular por ordenador la M.T. es una idea - congruente con la eliminación de tarea rutinaria y que permi-- te, dentro de ciertos límites, preservar y hasta acentuar los

aspectos teóricos de la M.T. Entre otras cosas permite -es un ejemplo- ayudar a poner a punto el diseño de una M.T. como la del apartado 2.3 (Delgado, 1.978).

Se han escrito algunos programas simuladores de M.T.'s (por ejemplo Curtis, 1.965) utilizando un lenguaje de progra-- mación* con las siguientes características:

DN : desplazamiento relativo de la cabeza N casillas a la derecha.

IN : desplazamiento relativo de la cabeza N casillas a la izquierda.

E(e): escritura de e en la casilla situada bajo la cabeza, con e $\in$ E.

T (α,e): transferencia condicional. Si el símbolo leido es e, se transfiere control a la instrucción de etiqueta α, si no el programa se desarrolla en secuencia. Cuando no figure símbolo se tratará de una transferencia incondi-- cional.

NOMBRE (A_1,A_2,....A_r): Nombre de un subprograma donde las A_k son etiquetas de instrucciones o símbolos de la M.T.. NOMBRE es la etiqueta de la -

---

* Existe en la E.T.S.I.T. un simulador de M.T., realizado bajo la dirección del autor, escrito en Fortran (Delgado, 1.978).

primera instrucción del - subprograma

END : corresponde al estado - - "stop".

Para la representación en el ordenador de los alfabetos - de la M.T. se emplearán los propios caracteres admitidos por - el ordenador y el único problema es el que se deriva de la limitación física del ordenador - respecto de la M.T.: la memoria del ordenador es finita y la memoria de la M.T. es infinita. - Por consiguiente, los topes que el programa simulador establezca en la memoria central del ordenador serán los que marquen - en cada simulación el espacio - de validez de ésta.

Veamos ahora, mediante un ejemplo, la forma en que habría que describir un esquema de M.T. con el lenguaje especificado - más arriba (anotaremos el blanco por una B). Tomemos un caso muy sencillo, como el de la M.T. que sumaba dos números enteros no nulos en el alfabeto {|}. - (Aquí se sustituirá el palote | por el 1, que es un símbolo admitido por cualquier ordenador).(Véase fig. 21).

```
M0   T(M3, 1)
     T(M4, B)
     T(M5, *)
M1   T(M6, 1)
     T(M7, B)
     T(M8, *)
M2   T(M9, 1)
     T(M10, B)
     T(M11, *)
M3   E(B)
     D1
     T(M2)
M4   D1
     T(M0)
M5   E(B)
     T(M20)
M6   I1
     T(M1)
M7   D1
     T(M0)
M8   I1
     T(M1)
M9   D1
     T(M2)
M10  E(1)
     T(M1)
M11  D1
     T(M2)
M20  END.
```

Fig. 21.

4.2. Máquina de Turing universal

Todas las M.T.'s pueden ser simuladas por otra máquina de Turing llamada M.T. universal, siempre que se le dé a ésta la información necesaria sobre la primera, a saber:

- contenido inicial de la cinta
- posición inicial de la cabeza
- estado inicial
- esquema funcional, programa o funciones f y g (como se prefiera llamar)

Los conceptos necesarios para definir una M.T. universal son éstos. Se demuestra que:

1°. *Cualquier M.T. concreta puede ser simulada por otra con alfabeto binario ({0,1} ; {□,|} o los dos símbolos preferidos de cada uno. En general, los símbolos de un alfabeto cualquiera son codificables por paquetes de unos, paquetes distinguibles entre sí por paquetes convenidos de ceros).*

2°. *Las posibilidades de una M.T. no se restringen por el hecho de que su cinta sea ilimitada sólo por un extremo.*

Una M.T. universal dispondrá de una cinta ilimitada por ambas partes, con un alfabeto externo {0,1,α,β, ≠, ≠,*,b} correspondiéndose α con 0 y β con 1. El símbolo * representa la posición de la cabeza simulada, b es el blanco y al iniciar y terminar la simulación de un movimiento de la M.T. simulada sólo habrá símbolos numéricos {0,1} en la M.T. universal. La información de datos de la cinta simulada se sitúa a partir de una posición a la izquierda de ≠ y la información sobre sus estados y funciones a la derecha de ese mismo símbolo. Esta es -

la versión de una M.T.U. recogida en (Scala, Minguet, 1.974).

Con estos elementos se puede construir una M.T. universal que, combinando las posibilidades de una M.T. de cinta direccionable (apartado 3) y de una M.T. transcriptora de información, puede simular todas las M.T.'s definidas en un alfabeto binario.

El lector interesado que quiera profundizar en este tema debe consultar la referencia en español (Scala, Minguet,1974) Un desarrollo un poco diferente, simple pero detallado, de la M.T. universal, se hallará en el capítulo 2 de (Hennie, 1.977).

4.3. Otras consideraciones.

Ya se ha visto que la propiedad de cualquier M.T. de contar con una cinta infinita -característica eminentemente teórica- le confiere una variedad (en el sentido de R. Ashby) adaptable potencialmente al control de cualquier circunstancia de tratamiento de información.

Es conocido que, cuando se recorre un terreno más práctico como es del uso de los ordenadores, el contar con una memoria principal grande es condición sine qua non para poder tratar,con una determinada velocidad, mayor variedad de problemas y problemas más complejos. Aumentando la capacidad de la memoria principal se incrementa el volumen y complejidad de los problemas que es posible tratar con una máquina concreta y a la inversa. Ahora bien, razones tecnológicas y económicas impiden aumentar todo lo que se desearía la capacidad de las memorias, para un precio y un instante histórico precisos. En todos los casos, la velocidad de la memoria representa un freno a la velocidad de que hace gala el procesador o unidad de cálculo que tiene la virtud, además, a cada nuevo diseño de orde-

nador, de poder ejecutar un repertorio más amplio de instrucciones distintas. Lo cierto es que, incluso con una memoria finita, el incremento del número de instrucciones distintas lo que hace es añadir versatilidad y velocidad al tratamiento de los problemas, porque un número extraordinariamente reducido de instrucciones distintas basta para describir cualquier cálculo en este tipo de máquinas.

El tipo de máquinas que definió Turing (antes que se diseñara el primer ordenador, recuérdese) es un ordenador ideal ya que no depende de ninguna característica física ni le preocupa la velocidad u otra clase de eficiencia. Visto desde la perspectiva de los ordenadores - esto es, a posteriori- *es un ordenador con una memoria infinita y una sola instrucción distinta*, la quíntupla, tantas veces utilizada en este capítulo. Así pues, no solamente goza de la virtud teórica de poder ejecutar cualquier algoritmo, como, por ejemplo, simular a un ordenador moderno que es una máquina más compleja dotada de un rico repertorio de instrucciones, sino que esto lo hace mediante los pasos más elementales de que se tiene noticia

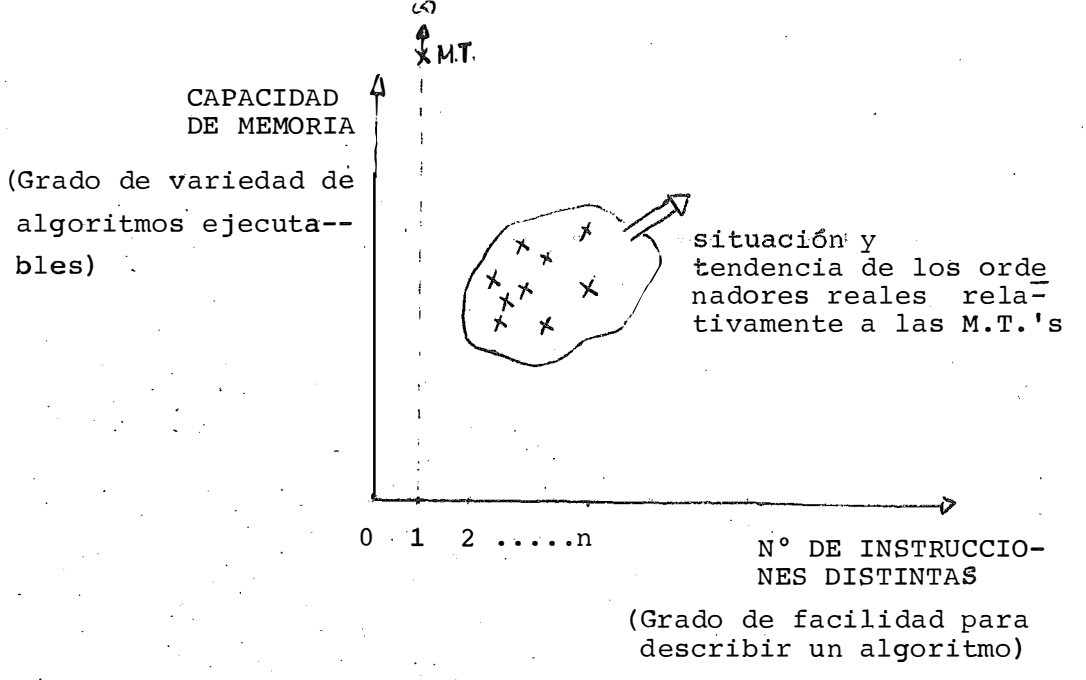


Fig. 22.

La genialidad de Turing consistió en poner su invento fuera de las limitaciones espacio-temporales (espacio para la información, infinito; tiempo, el que sea, pero un número finito de pasos). En tales circunstancias, una sola M.T. es capaz de reproducir el funcionamiento de todas las demás, siempre que disponga de la descripción de las mismas.

Esta última idea, inmanente a la M.T.U., de que una máquina pueda desarrollar procesos más complejos que los que su propia estructura parece permitirle, a condición de que se le suministre la información adecuada, despertó gran interés y ha sido trasladada por analogía al campo de la reproducción biológica para intentar explicar la construcción de la vida y su mantenimiento a partir de las informaciones genéticas (véase, p. ejemplo, Singh, 1.976, cap. 13).

5. Sucedaneos de la Máquina de Turing

Al objeto de que el lector tenga simplemente noticia de ello conviene tal vez que conozca la existencia de ciertos derivados de la versión clásica de M.T. que es con la que hemos venido trabajando en este capítulo. Son, entre otros:

a) M.T.'s con sólo dos de las tres salidas posibles e_k, m_h, q_p.

b) M.T.'s con cinta limitada por un extremo.

c) M.T.'s con más de una cinta.

d) M.T.'s no deterministas.

Se demuestra que ninguna de ellas restringe las posibilidades de la M.T. definida en el apartado 1 (Scala, Minguet, 1.974).

6. RESUMEN.

Una Máquina de Turing es un artefacto computador constituido por un autómata finito que controla una cinta infinita.- Cada paso en el cálculo de una M.T. consiste en escribir un - símbolo en la cinta, desplazar la cabeza de lectura/escritura un cuadro a la derecha o a la izquierda y asumir un nuevo estado. La acción concreta de cada paso viene determinada por el - estado en curso de la máquina y por el símbolo que lee en ese instante la cabeza.

El funcionamiento de una M.T. se especifica completa-- mente *por una lista de quíntuplas* e_i, q_j, e_k, m_h, q_p, donde están todas las combinaciones $e_i\ q_j$ que permiten los alfabetos exter no E e interno Q, *por la cinta con la información inicial* y - por la situación inicial de la máquina expresada *por la posición de la cabeza y el estado del autómata*. A la lista de quín tuplas se le llama esquema funcional o programa de la M.T.

Distintos ejercicios a lo largo del capítulo han busca do familiarizar al lector con el funcionamiento y las diferentes formas de representar los resultados de una M.T. Una M.T. es capaz de realizar sólo operaciones muy elementales, pero - secuencias adecuadas de estas operaciones pueden llegar a componer una amplia variedad de operaciones de manipulación de - bloques. Estas últimas operaciones comprenden: formar copias - de bloques especificados, sustituir un bloque por otro y compa rar bloques. Empleando estas operaciones como subprogramas, es posible diseñar M.T.'s que realizan cálculos muy complejos.

El modelo de M.T. que se ha presentado puede modificar se en varios sentidos sin alterar sus posibilidades últimas co mo máquina computadora. Tal vez convenga subrayar que, *dado un algoritmo a ejecutar por una máquina de Turing*, en el diseño -

de ésta -supuesto escogido un modelo específico de M.T. (por - ejemplo, con una sola cinta ilimitada por ambos extremos)- *se presenta, en principio, la disyuntiva de disminuir el cardinal del alfabeto externo a costa de aumentar el del interno, o viceversa.* Uno de los resultados más interesantes de esta propiedad es que siempre es posible simular una M.T. por otra M.T. - definida sobre un alfabeto externo binario.

Por último, debe resaltarse la Máquina de Turing universal, *diseñada para ejecutar un algoritmo de simulación de - todas las otras M.T.'s que poseen su misma estructura.* Las especificaciones completas de la M.T. simulada y sus datos de - trabajo figuran como datos en la cinta de la M.T.U.

CAPITULO 5.

MAQUINAS DE TURING: ALGORITMOS Y CALCULABILIDAD (RECURSIVIDAD)

0. INTRODUCCIÓN.

Este capítulo aborda de manera muy esquemática la noción de *calculabilidad* en el sentido de Turing (y conceptos relacionados), que formaliza la noción un tanto intuitiva de algoritmo de las definiciones de autores recogidas en el primer capítulo. Respecto a la definición formal de algoritmo del mismo primer capítulo ésta es una alternativa más fértil, pues se expresa en términos de una máquina que, no por ser ideal o conceptual, es menos concreta. Huelga decir que en esta máquina puede el lector comprender que además convergen los tres objetos del título de este tema: un algoritmo es representable por un programa (esquema funcional en una M.T.) al que, si es necesario, se llega por un proceso estructurado de refinamiento de grafos o de otras fórmulas.

1. FUNCIÓN CALCULABLE Y FUNCIÓN PARCIALMENTE CALCULABLE

1.1. Hipótesis de Turing.

Siguiendo a Arbib (Arbib, 1.965), la idea intuitiva de procedimiento efectivo para desarrollar un cálculo es la misma que la de algoritmo. Pues bien, según la hipótesis de Turing: *la noción intuitiva informal de un procedimiento efectivo sobre secuencias de símbolos es idéntica a la de nuestro concepto preciso de un procedimiento que puede ser ejecutado por una*

máquina de Turing.

No existe prueba formal de esta hipótesis pero, hasta la fecha, siempre que en la teoría de las *funciones recursivas** ha sido intuitivamente evidente que existía un algoritmo, ha sido posible diseñar una M.T. para ejecutarlo.

1.2. Función calculable

Asociamos una función $F_Z^{(r)}$ con una máquina de Turing Z, definiendo $F_Z^{(r)}$ $(n_1, n_2, \ldots . n_r)$ como el número $<\gamma_p>$ de unos que hay en la cinta cuando Z se detiene, habiéndose iniciado en la siguiente situación (véase para esta representación el apartado 2.1. del 4° capítulo):

$$.....000\underline{1}11...1011...11011...1101.....10111...1000... \quad (1)$$

$$n_1 + 1 \quad n_2 + 1 \quad n_3 + 1 \qquad\qquad n_r + 1$$

Si la máquina nunca se detuviera, $F_Z^{(r)}$ no estaría definida para la r- upla considerada.

Intentemos ver más de cerca el significado de los elementos que intervienen en esta función.

En el apartado 4.2 del capítulo anterior se estableció que un alfabeto binario es suficiente -a costa de aumentar el número de estados- para ejecutar cualquier cálculo con una M.T.

---

* *Funciones recursivas y funciones calculables son equivalentes. El concepto de calculabilidad se debe a Turing y el de recursividad a Kleene. (Gross, Lentin, 1.967).*

Con mayor razón podrán tratarse los números enteros no negativos, utilizando un alfabeto {0,1} o {□,|}. Escojamos la primera de estas dos representaciones.

Se representará un n° entero no negativo, n, por (n+1) unos y a este bloque lo llamaremos $\bar{n}$, para distinguirlo.

$$\bar{0} = 1,\ \bar{1} = 11,\ \bar{2} = 111,\ldots.,\ \bar{5} = 111111,\ldots.$$

El cero hará las veces de separador, por lo que una -r- upla $(n_1,n_2,\ldots.n_r)$ se convendrá en representar como en (1), lo que abreviadamente equivale a $\bar{n}_1 0 \bar{n}_2 0 \bar{n}_3 0 \ldots. 0 \bar{n}_r$. Si ésta es una información A en la cinta de una M.T.Z, Z será aplicable o no a A. En el primer caso, Z produce el cálculo

$$\gamma_1,\gamma_2,\ldots.\ \gamma_p, \quad \text{donde } \gamma_1 = q_0 \bar{n}_1 0 \bar{n}_2 0 \bar{n}_3 0 \ldots. 0 \bar{n}_r$$

y el n° entero $<\gamma_p>$ es una función que depende de la máquina Z y de la r- upla inicial. Se escribe

$$<\gamma_p> = F_Z^{(r)}\ (n_1,n_2,\ldots.n_r) \qquad (2)$$

que es una función de valores enteros no negativos, definida -sobre N^r o sobre una parte de N^r (función parcialmente definida).

Si se hace ahora al revés, es decir se parte de una función definida sobre N^r o sobre una parte de N^r se tienen las -siguientes definiciones.

1.2.1. Definición de función parcialmente calculable.

Se dice que una función f definida sobre una parte de N^r, es parcialmente calculable para expresar que existe una má

quina de Turing Z tal que, para toda r- upla a la que corres-- ponda un valor de f, se tenga:

$$f(n_1,n_2,.....,n_r) = F_Z^{(r)}\ (n_1,n_2,.....,n_r) \qquad (3)$$

1.2.2. Definición de función calculable

Se dice que una función f es calculable para expresar que está definida sobre N^r y es parcialmente calculable.

1.3. Ejemplos

1.3.1. La función $f(n_1,n_2) = n_1 + n_2$ definida sobre - las parejas de enteros no negativos, que es calculable en el - sentido habitual de la palabra, lo es también en el sentido de la definición 1.2.2. Puede construirse una M.T.Z tal que

$$n_1 + n_2 = SUM_Z^{(2)}\ (n_1,n_2) \qquad (4)$$

y teniendo en cuenta que $n_1 + n_2 = \bar{n}_1 + \bar{n}_2 - 2$

e_i \ q_j	q_0	q_1	q_2	q_3	q_4	q_5	q_6
0		$\rightarrow q_5$	q_4	1 $\leftarrow q_4$	$\rightarrow$ stop	$\leftarrow q_6$	
1	0 $\rightarrow q_1$	0 $\rightarrow q_2$	0 $\rightarrow q_3$	$\rightarrow$	$\leftarrow$	$\rightarrow$	0 $\leftarrow q_4$

En el momento de detenerse la máquina (4) contiene - - $(n_1 + n_2)$ unos, cualesquiera que sean n_1 y n_2. La función f es - definida sobre N^2 y es parcialmente calculable, luego es -

una función calculable.

1.3.2. La función $g(n_1,n_2) = n_1 \dot{-} n_2$, sustracción definida sobre el subconjunto $n_1 \geq n_2$ puede demostrarse que es parcialmente calculable sin más que construir la correspondiente M.T., situando, por ejemplo, el número n_1 a la izquierda y el n_2 a la derecha de un cero separador (véase (Gross, Lentin, 1.967), pag. 51).

La función $n_1 \dot{-} n_2$, parcialmente calculable, podría prolongarse en una función calculable $n_1 \dot{-} n_2$, definida sobre N^2, así:

$$n_1 \dot{-} n_2 = n_1 - n_2 \qquad \text{si } n_1 \geq n_2$$

$$n_1 \dot{-} n_2 = 0 \qquad \text{si } n_1 < n_2$$

2. NUMERABILIDAD DE LA COLECCIÓN DE TODAS LAS M.T.'S.

Una M.T. está especificada por una lista de quíntuplas $e_i\ q_j\ e_k\ m_h\ q_p$, que forman un conjunto finito. Los valores posibles de i, j, k, h, p son todos numerables. Así pues, la colección de todas las quíntuplas es numerable. Consiguientemente, las listas de quíntuplas son numerables y, por ende, los autómatas por ellas representados.

Esto significa que *las M.T.'S pueden ordenarse numéricamente*. El problema es cómo escoger un código tal que, dado un número, puedan determinarse las especificaciones de la M.T. correspondiente, si la hubiere, y viceversa.

2.1. Números de Gödel

El método de establecer un código de esta naturaleza, que consiste en *numerizar lo no numérico*, fué propuesto por Gödel antes de que existieran las máquinas de Turing.

Kurt Gödel, matemático desaparecido pocos meses antes de la primera redacción de este texto, escribió en 1.930 un artículo que cuando se publicó en una revista alemana* en 1.931, produjo el efecto de un paquete de dinamita colocado precisamente en la base de la viga maestra de los fundamentos de la matemática. Fundamentos que, con celo encomiable, estaban renovando los matemáticos de la época, con Hilbert a la cabeza.

Para el lector que no conozca en qué contexto propuso Gödel su técnica de codificación, la preocupación matemática del momento consistía en probar la consistencia de la teoría axiomática de conjuntos, para lo cual Hilbert propuso un programa completo. Pues bien, Gödel probó dos cosas:

1°. Si la teoría axiomática de conjuntos es consistente, existen teoremas que no pueden ser probados ni refutados.

2°. No existe ningún procedimiento constructivo que pruebe que la teoría axiomática de conjuntos es consistente.

El primer resultado prueba que los problemas no siem--

---

* Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. (Sobre proposiciones formalmente indecidibles de los Principia Mathematica y sistemas relacionados). Monatshefte für Mathematik und Physik 38, 1.931, pp. 173-98.

pre son solubles, ni siquiera en principio; el segundo destrozó el programa de Hilbert para probar la consistencia. (Stewart, 1.977).

En la teoría axiomática de conjuntos se utilizan símbolos con los que se forman expresiones o cadenas, que son objetos metamatemáticos. Para demostrar sus teoremas, Gödel se sirvió de una codificación numérica de dichas cadenas, que es el asunto que interesa aquí.

Supongamos que se cuenta con los símbolos que a continuación se reseñan. Cada uno de ellos es codificable por los números naturales en la manera indicada:

$$+ \quad - \quad x \quad \div \quad (\quad) \quad = \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad a \quad b \ldots x_i \ldots.$$

$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19 \ldots y_i \ldots.$$

Una de las formas más simples para representar cadenas arbitrarias $x_0, x_1, \ldots x_n$ compuestas con estos símbolos, por ejemplo, sería la de utilizar el número natural

$$p_0^{y_0} \; p_1^{y_1} \; \ldots . p_n^{y_n}$$

siendo p_i el i- esimo nº primo.

La fórmula

$$4 + 7 = 11$$

se codificaría por el número $2^{12}.3^{1}.5^{15}.7^{7}.11^{9}.13^{9}$

La fórmula

$$a(a - 1) = aa - a$$

tendría el siguiente número de Gödel: $2^{18}.3^5.5^{18}.7^2.11^9.13^6.17^7.19^{18}.23^{18}.29^2.31^{18}$.

La cadena $\div$ - + se codifica por $2^4.3^2.5^1$ igual a 720. Pero, a causa de la *unicidad de la factorización en números primos*, una cadena puede reconstruirse a partir de su código. Esto es, el número 720, factorizado, nos da $2^4.3^2.5^1$ que corresponde a la cadena $\div$ - +, pues los símbolos que componen ésta tienen los códigos 4, 2 y 1 respectivamente.

En resumen, *cada cadena tiene un número, y números diferentes corresponden a cadenas diferentes*. Disponiendo éstas según el tamaño de sus números de Gödel, puede verse que el conjunto de cadenas es numerable.

2.2. Catálogo de las M.T.'s.

Todas las quíntuplas que constituyen el esquema funcional de una máquina de Turing forman una cadena de símbolos codificable por un número de Gödel. O, lo que es lo mismo, cada M.T. recibe su número de Gödel z y con él puede catalogarse numéricamente la colección de las máquinas de Turing.

Tratándose de un procedimiento sistemático es posible diseñar M.T.'s con las siguientes propiedades:

a) *Existe una M.T. que es aplicable a toda secuencia escrita en su cinta (representativa de la lista de quíntuplas), transformándola en su número de Godel.*

b) *Existe una M.T. que, para todo número escrito en su cinta, proporciona una de estas dos respuestas:*

- *"no existe secuencia correspondiente a este número"*

- *"la secuencia correspondiente a este número es..."*
  (la lista de quíntuplas)

3. De nuevo, la máquina de Turing universal.

Se ha visto en el capítulo anterior que una M.T.U. puede simular a todas las otras si se le suministran los datos y las especificaciones de éstas. En el punto 2 acabamos de ver - que todas las máquinas pueden ser catalogadas unívocamente, y que puede diseñarse una M.T. tal que, dado un número de Gödel z, entrega como resultado la lista de quíntuplas de una M.T., si la hay asociada a ese número. Con esta última perspectiva, es factible pensar en una M.T.U. a la que baste suministrar solamente el número de catálogo de las M.T.'s que se quiera simular. ¿Qué tiene que ver con las funciones calculables?. Veámoslo.

Llámese X al elemento $(n_1, n_2, \ldots . n_r) \in N^r$. A toda función f(X) parcialmente calculable puede asociarsele el número de Gödel z de la M.T. que calcula dicha función (es evidente - que puede haber varias máquinas que calculen la misma función).

Se define una función $Q_z^{(r)}$:

- Si z es el número de Gödel de una máquina Z que calcula la función parcialmente calculable f(X), entonces $Q_z^{(r)}(X)$ coincide con f(X).

- Si z no es el número de Gödel de ninguna máquina, entonces $Q_z^{(r)}(X)$ toma el valor cero.

Está claro que $Q_z^{(r)}$ es una función cuyo dominio es - N^{r+1} y que toma sus valores en N. Puede calcularse por composi

ción de dos máquinas de Turing (véase apartado 2.3 del 4° capítulo).

A una M.T. como la definida en el apartado 2.2.b) se le suministra el número z. Si la respuesta es no, $Q_z^{(r)}$ toma el valor cero. Si la respuesta es sí, se alimenta otra M.T. con la secuencia producida por la anterior M.T. que será el programa de cálculo de Z, y con el elemento o dato X. El resultado de la ejecución de esta última máquina es $Q_z^{(r)}(X)$, igual a f(X).

Esto es lo mismo que decir que *existe una máquina de Turing universal* U, tal que*

$$F_Z(X) = F_U(z,X) \qquad (5)$$

para todas las M.T.'s Z y todos los enteros z.

3.1. Teorema

Las funciones $Q_z^{(r)}(X)$ son funciones parcialmente calculables.

Una M.T. que calcula la función $Q_z^{(r)}(X)$ se llama universal: inscribiendo en su cinta el número z adecuado, ella puede calcular la función parcialmente calculable correspondiente a este número.

---

* El análogo de U es el ordenador cargado de programas al que se le da el nombre z de uno de éstos y los datos sobre los que tiene que operar.

4. CONJUNTOS RECURSIVOS Y RECURSIVAMENTE NUMERABLES

En este apartado se va a considerar muy esquemáticamente la aplicación de los conceptos de calculabilidad a dos importantes clases de conjuntos de números naturales: los conjuntos recursivos y los conjuntos recursivamente numerables.

Hay dos procesos fundamentales de cálculo que pueden asociarse a un conjunto específico G de números naturales. Uno es el proceso de determinar si, dado cualquier número natural x, éste pertenece a G. El otro es el proceso de generar, uno a uno, todos los elementos de G. Ambos procesos están relacionados, pero no son equivalentes. (Hennie, 1.977).

Para expresar qué es un conjunto de alguna de estas clases debe recordarse previamente la definición de *función característica de un conjunto*. (Gross, Lentin, 1.967).

Sea $G \subset N$. La función característica de G, C_G, se define así:

$$C_G(x) = \begin{cases} 1, \text{ si } x \in G \\ 0, \text{ si } x \notin G \end{cases}$$

De la misma manera se define la función característica de un conjunto $G \subset N^r$.

4.1. Conjunto recursivo.

Decir que un conjunto es recursivo es expresar que su

función característica es calculable*.

Esta definición equivale obviamente a decir que existe una M.T. que, aplicada a la información X, la transforma en 1 o en 0. O, en otras palabras, que existe un procedimiento efectivo para decir si un elemento X pertenece o no a ese conjunto.

4.2. Conjunto recursivamente numerable.

Es el dominio de una (por lo menos) función parcialmente calculable*.

Dicho en otra forma, significa que existe un procedimiento efectivo para generar sus elementos, uno detrás de otro, (Arbib, 1.965).

Por ejemplo, el conjunto de los cuadrados de los números enteros es recursivamente numerable -se toman los números 1,2,3,4,.... y se van elevando al cuadrado sucesivamente. También es recursivo- dado un número entero cualquiera, se descompone en sus factores primos viéndose entonces con facilidad si es o no un cuadrado.

4.3. Dos teoremas más

Se demuestra que (Gross, Lentin, 1.967, pag.59), (Arbib, 1.965, pag. 21):

---

* Podría sustituirse "función calculable" por "función recursiva" (véase pie de página del apartado 1.1); y "función parcialmente calculable" por "función parcialmente recursiva".

4.3.1. Un conjunto es recursivo si y sólo si el mismo y su complementario son recursivamente numerables.

4.3.2. Existen conjuntos recursivamente numerables que no son recursivos.

5. Determinación de la finitud del proceso de cálculo. Problema de la aplicabilidad.

No podemos acabar este tema de "algoritmos, programación y máquinas de Turing" sin mencionar un famoso problema que se enuncia así (Scala, Minguet, 1.974, U.D.III, pag.XVII/4): Dada cualquier M.T., una cinta con un número finito de símbolos X y una posición inicial de la cabeza, establecer un algoritmo que permita conocer si el proceso se detendrá o no. "The halting problem" (el problema de la aplicabilidad), como se le conoce en la literatura, *es un problema indecidible.*

Bien, este problema puede enunciarse de una forma distinta. En el enunciado que se acaba de dar se habla de una máquina de Turing, supongamos que su nombre es una vez más Z con el número de orden z. En él se habla también de una algoritmo, así que según la hipótesis de Turing (apartado 1.1), que hemos aceptado, eso es lo mismo que hablar de una M.T. A esta nueva M.T. le llamaremos Z_H.

Nuevo enunciado, más general, del problema (Gross, Lentin, 1.967): ¿Existe una M.T. Z_H cuya información en cinta son parejas y que, cuando se le suministra la pareja (z,X), responde una de estas dos cosas:

a) Sí, la máquina Z, de número z, es aplicable al dato X.

b) No, la máquina Z, de número z, es inaplicable al dato X?.

Demostración de que el problema es indecidible

Supóngase que existe tal M.T., Z_H.

Sea E un conjunto recursivamente numerable no recursivo (véase teorema 4.3.2). E es el conjunto de definición de una función parcialmente definida calculada por la máquina Z_α.

Sea x un entero cualquiera, ante el que Z_H podría dar una respuesta así:

"Sí, $x \in E$, Z_α se parará"

"No, $x \in E$, Z_α no se parará"

Entonces E sería recursivo, contrariamente a la hipótesis. Q.e.d.

Observación.

El problema de la aplicabilidad es trasladable al terreno práctico de la programación de ordenadores. Dados un programa P y los datos D correspondientes ¿existe un programa general P_H que permita saber si P, aplicado a D, se detendrá?. La respuesta es que P_H no existe.

6. Las máquinas de Turing y los lenguajes tipo 0

La lingüistica formal ofrece un campo de interesante aplicación de las M.T.'s. Los distintos lenguajes formales se tratan con cierta extensión en este mismo volumen (tema 4), en

donde puede verse la clasificación de aquellos en niveles relacionados con las reglas gramaticales que los generan y con los mecanismos que los aceptan (autómatas de uno u otro tipo).

Por lo que se refiere al autómata especial llamado M.T. -asunto que concierne a los capítulos 4 y 5 de este tema- avancemos unas líneas de su relación con la lingüistica matemática.

- Puede construirse una M.T. que acepte cualquier lenguaje generado por un sistema de escritura no res- - tringido (lenguajes tipo 0).

- Cualquier lenguaje generado por una gramática del tipo 0 es recursivamente numerable.

Se dice que un lenguaje $L \subseteq E^*$ (véase tema 4, Cap.3., apart. 3) es recursivamente numerable si se puede construir - una M.T. que acepte todas las cadenas en L y ninguna otra. Un lenguaje es recursivo si tanto L como su complementario, E^*-L, son recursivamente numerables. Si L es recursivo se puede construir una M.T. (en la práctica, un programa) capaz de reconocer si una cadena dada es o no un elemento de L. Realmente, esta M.T. estaría compuesta de dos M.T.'s., M.T.1 y M.T.2. El - conjunto de ambas o M.T. principal se limitaría a anotar cual de las dos máquinas acepta la cadena (M.T.1 reconocería cadenas en L y M.T.2, cadenas en E^* - L).

Supóngase que L es recursivamente numerable, pero no - recursivo. En tal caso, no sería posible construir la M.T. - - principal a que se hacía referencia en el párrafo anterior, al no poderse construir la M.T.2. La consecuencia es que la M.T.1 resultaría insuficiente, ya que, de no pararse, no podría saberse si la máquina (en la práctica, el programa) se había encerrado en un bucle o, simplemente, necesitaba más tiempo para aceptar la cadena en cuestión.

Una línea lingüistica como la que acaba de esbozarse - se prolonga en importantes desarrollos en el campo de la Inteligencia Artificial (Hunt, 1.975, pag. 35).

7. RESUMEN.

Se han definido las funciones calculables como aquellas a las que corresponde una máquina de Turing aplicable a una - r- upla $\overline{n_1}0\overline{n_2}0\overline{n_3}....0\overline{n_r}$, que produce un número $\langle \gamma_p \rangle$ de unos.

Las M.T.'s pueden catalogarse mediante alguna *técnica de codificación*, tal que a cada máquina le corresponda un número entero positivo y a cada número entero positivo le corresponda como máximo una sola M.T. La técnica clásica, a nivel - teórico, es la de los *números z de Gödel*.

Así codificada la colección de M.T.'s, pueden diseñarse dos máquinas de Turing, una para codificar máquinas de Turing (es decir, transformar un esquema funcional en un número de Gödel) y otra para decodificar (es decir, transformar un número de Gödel en un esquema funcional de M.T., si ésta existe).

Con este instrumental se ha redefinido la máquina de - Turing Universal como aquella a la que sólo es necesario darle el número de catálogo de la M.T. a simular y los datos para ésta.

Las nociones de calculabilidad y de parcial calculabilidad se emplean en relación con los procesos de cálculo en - conjuntos de números naturales para definir qué es un conjunto recursivo (*función característica calculable*) y qué es un conjunto recursivamente numerable (*dominio de función parcialmente calculable*).

Por último, se han dado unas muestras de grado de interés de los conceptos de los *conjuntos recursivos y recursivamente numerables* aplicándolos al razonamiento acerca de la indecibilidad del problema de la aplicabilidad de cualquier Máquina de Turing y a la aceptación de lenguajes generados por sistemas de escritura no restringidos.

REFERENCIAS BIBLIOGRAFICAS

CAPITULO 1. ALGORITMOS.

ALABAU, A. y FIGUERAS, J. Algoritmos y máquinas. Universidad Politécnica de Barcelona, C.P.D.A., 1.975.

CORGE, CH. Eléments d'informatique. Larousse Université, 1.975.

KNUTH, D.E. Algoritmos. Investigación y Ciencia, nº9 Junio - 1.977, pp. 42-53.

STONE, H.S. Introduction to Computer Organization and Data Structures. McGraw-Hill, 1.972.

TRAKHTENBROT, B.A. Algoritmos. Perspectivas de la revolución de los computadores. Alianza Universidad, 1.975, pp 108-130. Edición inglesa: Prentice-Hall, 1.970.

WANG, H. Juegos, lógica y Computadores. Computadoras y Computación edit. R.R. Fenichel y J. Weizenbaum. Blume, 1.974, pp 150-158. Edición inglesa: W.H. Freeman. (El artículo de Wang se publicó en 1.965 en el Scientific American)

CAPITULOS 2 Y 3. PROGRAMACIÓN ESTRUCTURADA

ARBIB, M.A. Computers and the Cybernetic Society. Academic Press, 1.977. (Publicado en castellano por Editorial AC, bajo la supervisión de F. Sáez Vacas, 1.978).

BÖHM, C. y JACOPINI, G. Flow diagrams, Turing machines and languages with only two formation rules. Communications of the ACM, vol. 9, N° 5, Mayo 1.966.

DAHL, O.I., DIJKSTRA, E.W. y HOARE, C.A.R. Notes on structured programming. Academic Press, 1.972.

DATAMATION, Revolution in programming. Datamation, Dic. 1.973. Número dedicado preferentemente al tema de la programación estructurada.

GILB, T. Laws of unreliability. Datamation, vol. 21, N° 3, Mar. 1.975.

MADNICK, S.E. y DONOVAN, J.J. Operating Systems. McGraw-Hill, 1.974.

MILLS, H.D. The new math of computer programming. Communications of the ACM, vol. 18, N° 1, Enero 1.975.

REYERO, E., RODERO, S. y SAEZ VACAS, F. Racionalización de la programación. Documentación del curso diseñado y escrito bajo la dirección de F. Sáez Vacas. ERIA, 1.975.

SAEZ VACAS, F. Guía para un análisis estructurado de la programación estructurada. Inforprim 1.975. Proceso de Datos N° 54, En-Febr. 1.976.

TABOURIER, Y., ROCHFELD, A. y FRANK, C. La programmation structurée en informatique. Les Editions d'Organisation, 1.975.

TENNY, T. Structured Programming in Fortran. Datamation, Jul. 1.974.

WIRTH, N. Algorithms + Data Structures = Programs. Prentice-Hall 1.976.

CAPITULOS 4 Y 5. MÁQUINAS DE TURING.

ARBIB, M.A. Brains, machines and mathematics. McGraw-Hill paperbacks, 1.965. (Existe versión castellana en la colección Alianza Universidad).

CORGE, CH. Eléments d'informatique. Larousse Université, 1.975.

CURTIS, M.W. A Turing machine simulator. Journal of the A.C.M. Vol. 12, N° 1, En. 1.965. pp 1-13.

DELGADO KLOOS, C. Simulador de Máquina de Turing. Proyecto Fin de Carrera, Septiembre 1.978.

GROSS, M. y LENTIN, A. Notions sur les grammaires formelles. Gauthier Villars, 1.967. (Existe versión castellana en la colección Tecnos).

HENNIE, F. Introduction to computability. Addison-Wesley, 1.977.

HUNT, E.B. Artificial Intelligence. Academic Press, 1.975.

SCALA, J.J. y MINGUET, J.M. Informática II. Unidad didáctica 3. Universidad Nacional de Educación a Distancia, 1.974.

SINGH, J. Teoría de la información, del lenguaje y de la cibernética. Alianza Universidad, 2ª. edición, 1.976.

STEWART, I. Conceptos de matemática moderna. Alianza Universidad, 1.977.

WANG, H. Juegos, lógica y computadores. (Véase la bibliografía sobre Algoritmos).

APENDICE

SIMULADOR DE MAQUINA DE TURING

Presentamos a continuación dos ejemplos de simulación de máquinas de Turing (Delgado,1978), cuyo esquema y resultados podrá cotejar el lector con los de los apartados 2.2 y 2.3 (M.T.3). El convenio utilizado en la impresión con el ordenador para el que se ha programado el simulador es el siguiente:

O	significa	□
I	"	|
>	"	un paso a la derecha
=	"	quieta
<	"	un paso a la izquierda
Λ	"	posición de la cabeza.

El primer ejemplo es una máquina de Turing que calcula el máximo común divisor de dos números naturales no nulos expresados en forma de palotes (apartado 2.2). Los resultados recogidos en las páginas 134 y 135 son el esquema funcional y el contenido de la cinta en los instantes en que se produce un cambio de estado. Los números escogidos fueron el 4 y el 6.

El segundo ejemplo es una máquina de Turing que convierte un número N expresado en {|} al alfabeto D. El programa simulador se ha aplicado sucesivamente a los casos numéricos: a) N = 14; b) N = 71; c) N = 1024. En todos los casos se ha escrito solamente el contenido inicial y final de la cinta, y en el último de ellos el programa simulador tardó en ejecutarse completamente 17 minutos y 20 segundos, necesitando 1.051.875 pasos la M.T. simulada.

MAQUINA DE TURING - ALGORITMO DE EUCLIDES EN {I}

MAQUINA DE TURING

(ALFABETO INTERNO) 5 ESTADOS: DE Q0 A Q4
(ALFABETO EXTERNO) 5 SIMBOLOS: 0I*AB

(ESQUEMA FUNCIONAL)

---ENTRADAS---		-------SALIDAS-------		
SIMBOLO	ESTADO	MOVIM.	SIMBOLO	ESTADO
0	Q0	>	0	Q3
0	Q1	>	0	Q4
0	Q2	<	0	Q3
0	Q3	>	0	Q1
0	Q4	=	0	STOP
I	Q0	>	B	Q0
I	Q1	=	A	Q2
I	Q2	=	B	Q1
I	Q3	>	I	Q1
I	Q4	<	I	Q1
*	Q0	<	A	Q0
*	Q1			
*	Q2	>	I	Q2
*	Q3	>	0	Q2
*	Q4			
A	Q0			
A	Q1	<	A	Q1
A	Q2	>	A	Q2
A	Q3	<	I	Q3
A	Q4	>	0	Q4
B	Q0	<	*	Q0
B	Q1	<	B	Q1
B	Q2	>	B	Q2
B	Q3	<	B	Q3
B	Q4	>	I	Q4

SE IMPRIME EL CONTENIDO DE LA CINTA
EN CADA CAMBIO DE ESTADO ENTRE EL PASO 30 Y EL

CINTA ESTADO Q0 SITUACION INICIAL
III*III
^

CINTA ESTADO Q1 PASO 34
000AAA0IIII
^

CINTA ESTADO Q2 PASO 41
000AAA0IIII
^

CINTA ESTADO Q1 PASO 49
000AAA0III
^

```
CINTA                ESTADO   Q4        PASO    58
QQAAAABBBBII
 !^

CINTA                ESTADO   Q1        PASO    67
QQQQQQIIIIII
 !     ^

CINTA                ESTADO   Q2        PASO    68
QQQQQQIIIAII
 !     ^

CINTA                ESTADO   Q1        PASO    70
QQQQQQIIIABI
 !     ^

CINTA                ESTADO   Q2        PASO    73
QQQQQQIIAABI
 !     ^

CINTA                ESTADO   Q1        PASO    77
QQQQQQIIAABB
 !     ^

CINTA                ESTADO   Q2        PASO    82
QQQQQQIAAABB
 !     ^

CINTA                ESTADO   Q3        PASO    88
QQQQQQIAAABBQ
 !     ^

CINTA                ESTADO   Q1        PASO    94
QQQQQQIIIQQ●
 !     ^

CINTA                ESTADO   Q2        PASO    95
QQQQQQIAIIQQQ
 !     ^

CINTA                ESTADO   Q1        PASO    97
QQQQQQIABIQQQ
 !     ^

CINTA                ESTADO   Q2        PASO   100
QQQQQQAABIQQQ
 !     ^

CINTA                ESTADO   STOP      PASO   114
QQQQQQQQIIQQQ
 !     ^
```

MAQUINA DE TURING - CONVERSION DE N ∈ {|} A N ∈ D.

MAQUINA DE TURING

ALFABETO INTERNO) 3 ESTADOS: DE Q0 A Q2
ALFABETO EXTERNO) 12 SIMBOLOS: 0|0123456789

ESQUEMA FUNCIONAL)

---ENTRADAS---		-------SALIDAS-------		
SIMBOLO	ESTADO	MOVIM.	SIMBOLO	ESTADO
0	Q0	=	1	Q2
0	Q1	=	0	STOP
0	Q2	<	0	Q1
1	Q0	<	1	Q0
1	Q1	<	0	Q0
1	Q2	>	1	Q2
	Q0	=	1	Q2
0	Q1	=	0	STOP
0	Q2	>	0	Q2
1	Q0	=	2	Q2
1	Q1	=	1	STOP
1	Q2	>	1	Q2
2	Q0	=	3	Q2
2	Q1	=	2	STOP
2	Q2	>	2	Q2
3	Q0	=	4	Q2
3	Q1	=	3	STOP
3	Q2	>	3	Q2
4	Q0	=	5	Q2
4	Q1	=	4	STOP
4	Q2	>	4	Q2
5	Q0	=	6	Q2
5	Q1	=	5	STOP
5	Q2	>	5	Q2
6	Q0	=	7	Q2
6	Q1	=	6	STOP
6	Q2	>	6	Q2
7	Q0	=	8	Q2
7	Q1	=	7	STOP
7	Q2	>	7	Q2
8	Q0	=	9	Q2
8	Q1	=	8	STOP
8	Q2	>	8	Q2
9	Q0	<	0	Q0
9	Q1	=	9	STOP
9	Q2	>	9	Q2

a) N = 14

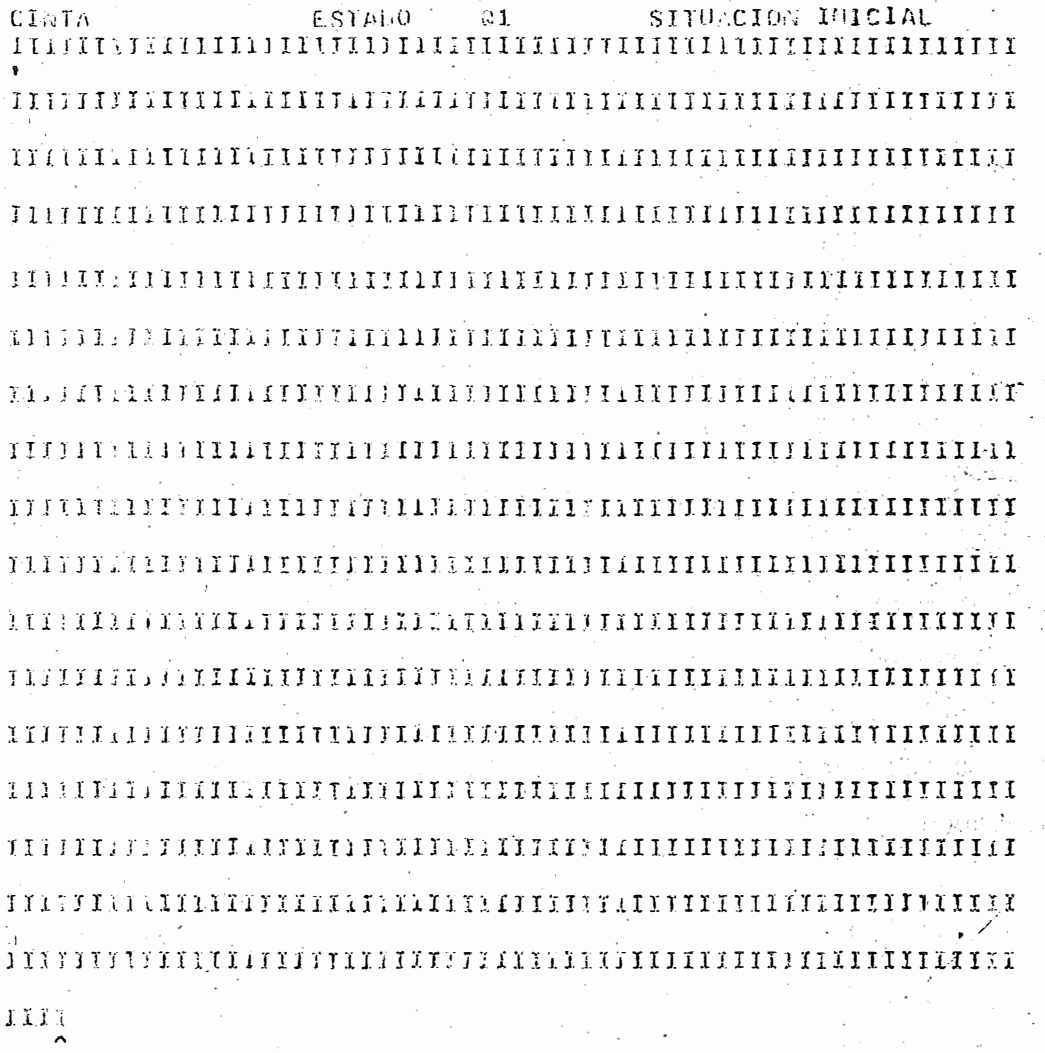
```
CINTA                ESTADO    Q1        SITUACION INICIAL
|||||||||||||||
 ^

CINTA                ESTADO    STOP      PASO     241
1406000000000000
 ^
```

b) N = 71

```
CINTA                ESTADO    Q1        SITUACION INICIAL
|||||||||||||||
```

c) N = 1024



CINTA ESTADO STOP PASO 1051875

1024

TEMA 4

LENGUAJES

GREGORIO FERNANDEZ FERNANDEZ

INDICE

PÁG.

Pág.

Pág.

Pág.

CAPITULO 1.

IDEAS GENERALES

1. ORIGEN DE LA TEORÍA DE LENGUAJES FORMALES.

La teoría de lenguajes formales nace como consecuencia de los estudios en un campo inicialmente bastante alejado de - la informática: la lingüística. En lingüística es tradicional distinguir entre *gramática particular* (propiedades particula-- res de lenguajes concretos, obtenidas fundamentalmente por pro cedimientos estadísticos) y *gramática universal* (propiedades - generales que pueden aplicarse a cualquier lenguaje humano) - (CHOMSKY, 1.967).

Los lingüistas de la escuela estructuralista americana habían desarrollado por los años 50 una serie de conceptos básicos relacionados con la gramática universal. Por ejemplo, si un lenguaje es un conjunto potencialmente infinito de frases, para describir un lenguaje deberá darse una *gramática generativa* o conjunto de reglas que subyacen en la composición de frases correctas y una *descripción estructural* de cada frase que permita explicar cómo tal frase puede componerse a partir de - una gramática dada. Fue Noam Chomsky quien, en una serie de publicaciones aparecidas a partir de 1.956, formalizó estos conceptos, sentando las bases de la moderna *lingüística algebráica*.

2. INTERÉS DE LOS LENGUAJES FORMALES EN INFORMÁTICA.

Pocos años después de los desarrollos iniciales de -

Chomsky, el concepto de gramática formal adquirió gran importancia en el estudio de los lenguajes de programación cuando ALGOL 60 (con algunas modificaciones sobre su versión original) pudo ser definido mediante una gramática de contexto libre (uno de los tipos de gramática definidos por Chomsky). Esto condujo de una manera natural a procedimientos de compilación orientada por sintaxis y al concepto de compilador de compiladores, puntos éstos que tocaremos en el Capítulo 5.

Desde una perspectiva bastante ambiciosa, hace tiempo se piensa que, puesto que todos los lenguajes de programación son lenguajes naturales susceptibles de ser formalizados, la teoría de lenguajes formales podría ser el fundamento de una teoría general de la programación, hoy inexistente (HARRISON, 1.965).

Por otra parte, a partir de su vertiente de aplicación a la lingüística, los lenguajes formales son de gran utilidad para el trabajo en dos campos situados en lo que se suele llamar "inteligencia artificial": la traducción automática y el procesamiento de lenguajes naturales.

3. OBSERVACIÓN SOBRE LA TERMINOLOGÍA.

En el Tema "Autómatas" hemos definido un lenguaje L como un subconjunto del lenguaje universal E^*, siendo los elementos de éste las infinitas *cadenas* que pueden formarse a partir de un conjunto finito de símbolos, E, al que llamábamos *alfabeto*. Esta es la terminología habitual en las publicaciones sobre teoría de autómatas (aunque algunos utilizan "palabra" y otros "secuencia" en lugar de "cadena"), y es la que continuaremos utilizando en este Tema.

Ahora bien, en los trabajos de orientación más lingüis

ta, al conjunto de base E se le suele llamar *vocabulario* (y - representar por V) y a los elementos de un lenguaje *sentencias o frases*. Nosotros utilizaremos "sentencia" en un sentido que precisaremos en el siguiente Capítulo para designar a las cadenas que pertenecen a un determinado lenguaje, a diferencia de "cadena" en general, que será un elemento cualquiera de E^*.

Según la aplicación, esta terminología puede variar. - Por ejemplo, en una teoría de la programación en ensamblador, E podría ser el repertorio de instrucciones, conjunto de etiquetas, etc., $x \in L$ sería un programa correcto, y $x \in E^* - L$ sería un programa incorrecto.

4. REPRESENTACIÓN DE LOS LENGUAJES.

Si un lenguaje es finito, su representación es inmediata: basta enumerar las cadenas que lo forman. Pero si es infinito, tendremos que dar una descripción finita; esta descripción será a su vez una cadena de símbolos combinados de acuerdo con ciertas reglas (sintaxis) y con un determinado significado tanto para los símbolos como para las reglas (semántica). Por tanto, esta cadena de símbolos pertenecerá a un *metalenguaje* que sirve para describir a nuestro lenguaje. Por ejemplo, - una expresión regular es una cadena del lenguaje de las expresiones regulares, que es un metalenguaje para describir a los lenguajes regulares.

Ahora bien, podemos preguntarnos si, dado un lenguaje infinito cualquiera, $L \subset E^*$, podemos siempre encontrar una representación finita. La contestación es "no". En efecto, sabemos (Tema "Algoritmos", Cap.5, Ap. 2.1) que E^* (conjunto de todas las cadenas sobre un alfabeto E) es recursivamente numerable - (a cada cadena se le puede asociar, por ejemplo, su número de Gödel) y, por tanto, cualquier $L \subset E^*$ también lo será. Suponga--

mos que existe un metalenguaje $L_M \subset E_M^*$ tal que cada cadena de L_M es una representación finita de un lenguaje sobre E^*, es decir, tal que a cada $L(M) \subset E^*$ corresponda al menos un $x_M \in L_M$. - Si tal metalenguaje existiera, debería ser, como todo lenguaje, recursivamente numerable. Esto implicaría que el conjunto de - todos los $L \subset E^*$ sería recursivamente numerable. Sin embargo, - existe un teorema en teoría de conjuntos que dice que el con-- junto de los subconjuntos de un conjunto recursivamente numerable (y E^* lo es) no es recursivamente numerable, lo que contradice la conclusión anterior y destruye la hipótesis de que - - exista un L_M para todo $L \subset E^*$. Por consiguiente, *no todos los posibles lenguajes tienen una representación finita en un meta-- lenguaje.*

En este Tema veremos algunos métodos para representar ciertas clases de lenguajes desde dos puntos de vista: el *generativo* (algoritmos que permitan generar todas las cadenas del lenguaje) y el *de reconocimiento* (algoritmos que permitan determinar si $x \in L$ o $x \notin L$).

5. Gramáticas, algoritmos y máquinas de Turing.

Una gramática es un algoritmo enumerativo que permite generar las cadenas de un lenguaje. Así, la gramática más general puede expresarse en términos de una máquina de Turing.

Decir que la gramática G de un lenguaje L(G) es un algoritmo enumerativo equivale a decir que existe una M.T. que - hace corresponder al número 1 la cadena x_1, al 2 x_2, etc., de manera que produce el conjunto recursivamente numerable - - $L(G) = \{x_1, x_2, \ldots\}$.

En relación con los lenguajes naturales, es interesante destacar que la introducción de gramáticas generativas, for

malizadas gracias a los desarrollos matemáticos en algoritmos y computabilidad, ha permitido explicar su *carácter creativo*, es decir, el hecho de que el lenguaje natural tiene mecanismos recursivos que le permiten expresar un número potencialmente infinito de ideas, sentimientos, etc. La falta de un formalismo para estudiar estos mecanismos condujo a ciertos lingüistas de orientación conductista a negar esta propiedad, y a otros, como Saussure, a considerarla como algo ajeno al campo de la lingüística (CHOMSKY, 1.967).

6. Dos ejemplos.

Los ejemplos que desarrollamos a continuación nos servirán para introducir de una manera intuitiva algunos de los conceptos que formalizaremos en Capítulos posteriores.

Puesto que aún no hemos definido lo que es una gramática, haremos uso de una que, mas o menos vagamente, todos conocemos: la gramática del lenguaje castellano.

Para describir la formación de sentencias (oraciones) utilizaremos *categorías sintácticas* o *sintagmas*, como "nombre", "verbo", etc. (o sintagma nominal, sintagma verbal, etc.). Está claro que, si bien "nombre" es un nombre, "verbo" no es un verbo. Esto es debido a que para describir el lenguaje castellano utilizamos como metalenguaje el propio castellano, y ponemos las comillas para destacar que la palabra en cuestión se utiliza como elemento del metalenguaje (es la diferencia que se hace en lógica entre *uso* y *mención* del lenguaje, ver Tema "Lógica", Cap. 1, Ap. 2). La notación habitualmente seguida no es poner comillas, sino encerrar la palabra entre paréntesis angulares: <nombre>, <verbo>, etc.

Una sentencia castellana sintácticamente correcta esta

rá compuesta por palabras concretas pertenecientes a las diversas categorias sintácticas (uno o varios sintagmas nominales, uno o varios sintagmas predicativos, etc.) combinadas de acuerdo con ciertas reglas.

Por ejemplo, una regla puede ser:

(1) *Para formar una sentencia, póngase un sintagma nominal y a continuación un sintagma predicativo.*

Y otras:

(2) *Un nombre propio es un sintagma nominal*

(3) *"España" es un nombre propio.*

(4) *Un sintagma predicativo puede formarse con una forma verbal transitiva y un sintagma nominal.*

(5) *"Es" es una forma verbal transitiva*

(6) *Un sintagma nominal puede formarse con un determinante y un nombre común.*

(7) *Un artículo es un determinante.*

(8) *"Un" es un artículo.*

(9) *"Estado" es un nombre común.*

Del conjunto de estas nueve reglas puede deducirse que

España es un estado

es una sentencia del castellano.

Las reglas anteriores pueden expresarse formalmente así:

(1) <Sentencia> → <SN> <SP>

(2) <SN> → <Npr>

(3) <Npr> → España

(4) <SP> → <FVtr> <SN>

(5) <FVtr> → es

(6) <SN> → <Det> <Ncom>

(7) <Det> → <Art>

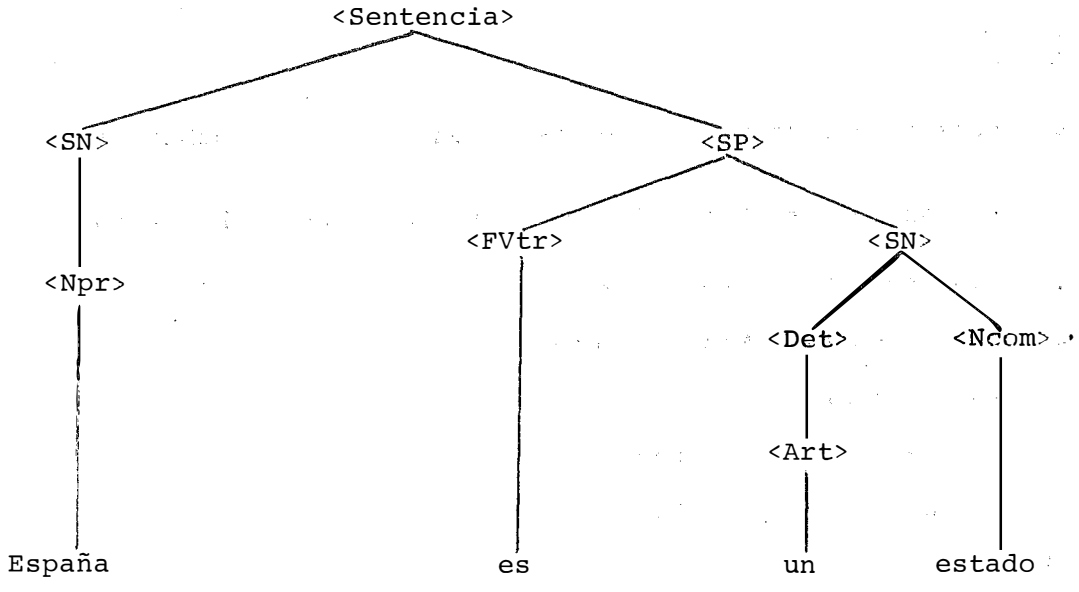
(8) <Art> → un

(9) <Ncom> → estado.

Y la sentencia obtenida puede descomponerse sintácticamente de acuerdo con las reglas; esta descomposición puede indicarse mediante corchetes etiquetados:

```
[  [España] ] [  [ es ] [  [    [ un] ]  [estado] ] ]
SN  Npr       SP  FVtr   SN  Det   Art      Ncom
```

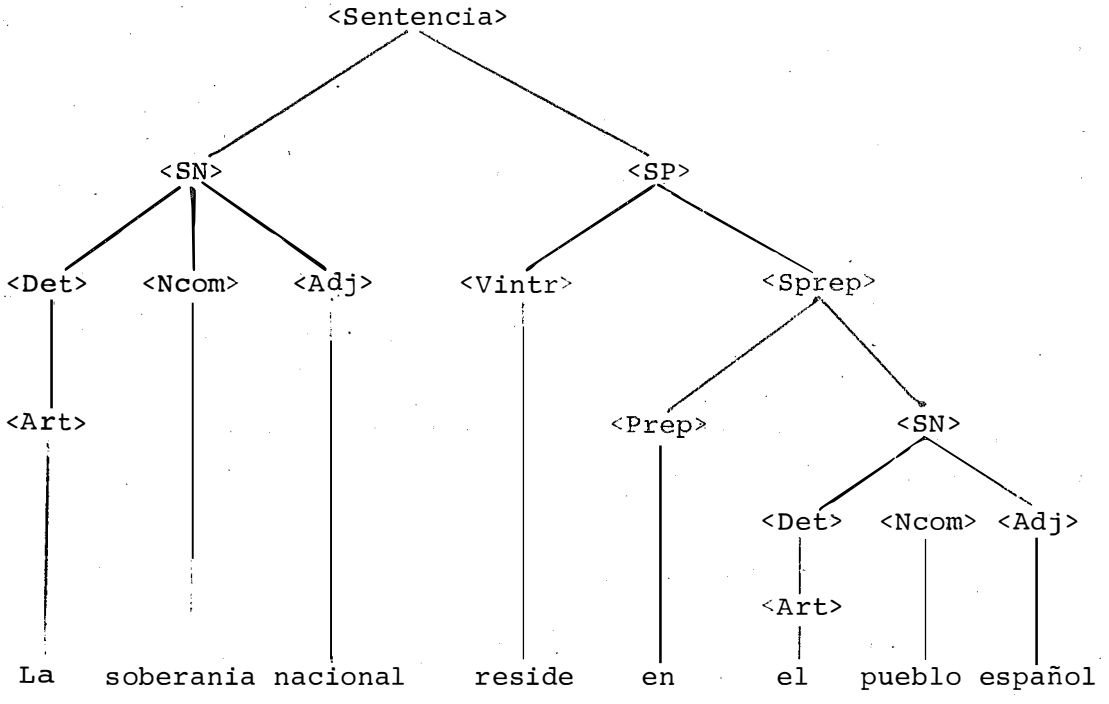
o bien, de una forma más gráfica, con un árbol llamado *árbol de derivación* o *árbol sintáctico*:



Como segundo ejemplo, consideremos la sentencia

La soberanía nacional reside en el pueblo español.

Su descomposición sintáctica, o derivación a partir de las reglas, es la indicada por el siguiente árbol:



Las reglas utilizadas para esta derivación han sido:

(1) <Sentencia> → <SN> <SP>

(2) <SN> → <Det> <Ncom> <Adj>

(3) <Det> → <Art>

(4) <SP> → <Vintr> <Sprep>

(5) <Sprep> → <Prep> <SN>

(6) <Art> → la

(7) <Art> → el

(8) <Ncom> → soberanía

(9) <Ncom> → pueblo

(10) <Adj> → nacional

(11) <Adj> → español

(12) <Vintr> → reside

(13) <Prep> → en

Se obtiene una sentencia partiendo de la regla (1) y aplicando las demás hasta que resulta una cadena con sólo *símbolos terminales*, en este caso, las palabras concretas del castellano obtenidas por aplicación de las *reglas terminales* (6) a (13).

Con estas reglas pueden derivarse otras sentencias, como

"El pueblo español reside en la soberanía nacional",

"La pueblo nacional reside en la pueblo español",

etc., que serían sentencias correctas en la gramática definida por tales reglas. (Existen precedimientos en lingüística para evitar que puedan derivarse sentencias carentes de sentido).

CAPITULO 2.

FUNDAMENTOS DE LA TEORIA DE LENGUAJES FORMALES

1. DEFINICIÓN DE GRAMÁTICA

Una gramática es una cuádrupla

$$G = \langle E_T, E_A, P, S \rangle,$$

donde

E_T es un conjunto finito llamado alfabeto principal o alfabeto de símbolos terminales

E_A es un conjunto finito llamado alfabeto auxiliar o alfabeto de variables.

P es un conjunto finito de pares ordenados (α,β), donde $\alpha \in (E_T \cup E_A)^+$ y $\beta \in (E_T \cup E_A)^*$. Estos pares se llaman reglas de escritura o producciones, y generalmente se escriben con la notación $\alpha \to \beta$. α es el antecedente de la regla y β el consecuente. Si $\beta \in E_T^*$ se dice de la regla que es una regla terminal.

$S \in E_A$ es un símbolo destacado del alfabeto auxiliar llamado símbolo inicial.

Llamaremos $E = E_T \cup E_A$; supondremos que $E_T \cap E_A = \emptyset$, y para distinguir los elementos de los diferentes conjuntos adoptamos los siguientes convenios:

Letras mayúsculas del alfabeto latino para los elemen-

tos de E_A. (O bien, palabras del metalenguaje - castellano entre paréntesis angulares).

Letras minúsculas del comienzo del alfabeto latino (a, b, c, ...), o cifras, para los elementos de E_T. (O bien, palabras del castellano).

Letras minúsculas del final del alfabeto latino (w, x, y, z) para los elementos de E_T^*.

Letras minúsculas del alfabeto griego para los elementos de E^+.

2. Relaciones entre cadenas de E^*

2.1. Relación de derivación directa, $\underset{G}{\Rightarrow}$

Si $(\alpha \rightarrow \beta) \in P$ y $\gamma, \delta \in E^*$, las cadenas $\gamma\alpha\delta$ y $\gamma\beta\delta$ están en la relación de derivación directa en la gramática G. Escribiremos entonces

$$\gamma\alpha\delta \underset{G}{\Rightarrow} \gamma\beta\delta$$

y diremos que la cadena $\gamma\beta\delta$ deriva directamente de la $\gamma\alpha\delta$, o - bien que $\gamma\alpha\delta$ produce directamente $\gamma\beta\delta$ en la gramática G. (De - ahí el nombre de producciones para los elementos de P).

2.2. Relación de derivación, $\overset{*}{\underset{G}{\Rightarrow}}$

Dados α_1, $\alpha_m \in E^*$, diremos que están en la relación de derivación en la gramática G si existen α_2, α_3..... α_{m-1} tales que

$$\alpha_1 \underset{G}{\Rightarrow} \alpha_2; \alpha_2 \underset{G}{\Rightarrow} \alpha_3; \ldots\ldots \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m$$

Se escribirá entonces

$$\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m,$$

diciendo que α_m deriva de α_1, o que α_1 produce α_m

Por convenio, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha \quad \forall \alpha \in E^*$

Siempre que sea evidente que nos referimos a una determinada gramática, G, escribiremos $\Rightarrow$ y $\overset{*}{\Rightarrow}$ en lugar de $\underset{G}{\Rightarrow}$ y $\overset{*}{\underset{G}{\Rightarrow}}$

Obsérvese que ni $\Rightarrow$, ni $\overset{*}{\Rightarrow}$, son relaciones de equivalencia, ya que, en general, no son simétricas. Por otra parte, $\overset{*}{\Rightarrow}$ es el cierre transitivo de $\Rightarrow$.

3. Lenguaje generado por una gramática. Equivalencia de gramáticas.

Una cadena $\xi \in E^*$ es una forma sentencial de la gramática G si existe una derivación que produce ξ a partir de S, - es decir, si $S \overset{*}{\underset{G}{\Rightarrow}} \xi$. Si además ξ sólo contiene símbolos terminales entonces es una sentencia o cadena válida. El conjunto - de sentencias que pueden generarse en una gramática G se llama lenguaje generado por la gramática G es decir:

$$L(G) = \{x \mid x \in E_T^* \text{ y } S \overset{*}{\underset{G}{\Rightarrow}} x\}$$

Dos gramáticas, G_1 y G_2, son equivalentes si ambas generan el mismo lenguaje: $L(G_1) = L(G_2)$.

4. EJEMPLOS.

4.1. Ejemplo 1.

Sea la gramática definida por $E_T = \{0,1\}$; $E_A = \{S\}$;

$$P = \{(S \rightarrow 000\ S\ 111);\ (0\ S\ 1 \rightarrow 01)\}$$

La única forma de generar sentencias es aplicando cualquier número de veces la primera regla y terminado con una aplicación de la segunda:

$$S \Rightarrow 000\ S\ 111 \Rightarrow 000000\ S\ 111111 \Rightarrow .... \Rightarrow 0^{3n-1}0S11^{3n-1} \Rightarrow 0^{3n}\ 1^{3n}$$

Por consiguiente, el lenguaje generado es el conjunto infinito

$$L(G) = \{0^{3n}\ 1^{3n} \mid n \geq 1\}$$

Si la segunda regla fuera $S \rightarrow 01$, el lenguaje sería

$$L(G) = \{0^{3n+1}\ 1^{3n+1} \mid n \geq 0\}$$

Como podrá comprobarse en otros ejemplos, no siempre es posible expresar de esa manera (por intensión) L(G).

4.2. Ejemplo 2.

$$E_A = \{S, A\};\quad E_T = \{a,b\}$$

$$P = \{(S \rightarrow abAS);\quad (abA \rightarrow baab);\ S \rightarrow a;\quad A \rightarrow b\}$$

Aquí, L(G) está compuesto por cadenas que contienen (abb) y (baab) intercambiándose y reproduciéndose cualquier número de veces, y terminando siempre la cadena con el símbolo a.

4.3. Ejemplo 3.

$$E_A = \{S, A, B\} \; ; \quad E_T = \{a,b\}$$

$$P = \left\{ \begin{array}{ll} S \to aB & A \to bAA \\ S \to bA & B \to b \\ A \to a & B \to bS \\ A \to aS & B \to aBB \end{array} \right\}$$

El lenguaje generado es el conjunto de todas las cadenas de E_T^+ que tienen igual número de a que de b, pero la demostración en este caso no es tan inmediata (HOPCROFT y ULLMAN, - 1969, pág. 13).

4.4. Ejemplo 4.

$$E_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$E_A = \{\langle \text{número} \rangle \; , \; \langle \text{cifra} \rangle\}$$

$$S = \langle \text{número} \rangle$$

$$P = \left\{ \begin{array}{lll} \langle \text{número} \rangle \to \langle \text{cifra} \rangle \; \langle \text{número} \rangle & & (1) \\ \langle \text{número} \rangle \to \langle \text{cifra} \rangle & & (2) \\ \langle \text{cifra} \rangle \to 0 & & (3) \\ \langle \text{cifra} \rangle \to 1 & & (4) \\ \langle \text{cifra} \rangle \to 2 & & (5) \\ \langle \text{cifra} \rangle \to 3 & & (6) \\ \langle \text{cifra} \rangle \to 4 & & (7) \\ \langle \text{cifra} \rangle \to 5 & & (8) \\ \langle \text{cifra} \rangle \to 6 & & (9) \\ \langle \text{cifra} \rangle \to 7 & & (10) \\ \langle \text{cifra} \rangle \to 8 & & (11) \\ \langle \text{cifra} \rangle \to 9 & & (12) \end{array} \right\}$$

Damos a continuación algunos ejemplos de derivaciones

de sentencias, poniendo bajo el símbolo $\Rightarrow$ el número de la regla o reglas utilizadas:

<número> $\overset{2}{\Rightarrow}$ <cifra> $\overset{3}{\Rightarrow}$ 0

<número> $\overset{1}{\Rightarrow}$ <cifra> <número> $\overset{12}{\Rightarrow}$ 9 <número> $\overset{2}{\Rightarrow}$ 9 <cifra> $\overset{12}{\Rightarrow}$ 99

<número> $\overset{1}{\Rightarrow}$ <cifra> <número> $\overset{1}{\Rightarrow}$ <cifra> <cifra> <número> $\overset{1}{\Rightarrow}$

<cifra> <cifra><cifra> <número> $\overset{2}{\Rightarrow}$ <cifra> <cifra>

<cifra> <cifra> $\overset{*}{\underset{4,5,6,7}{\Longrightarrow}}$ 1234

Como los símbolos terminales son las diez cifras decimales, el lenguaje que se obtiene es el conjunto infinito de cadenas que representan en decimal a los números naturales.

Obsérvese que es la regla (1) la que permite obtener una cadena de cifras de cualquier longitud.

4.5. Ejemplo 5.

$$E_T = \{a,b\}; \quad E_A = \{A,S\};$$

$$P = \left\{ \begin{array}{ll} S \to aS & (1) \\ S \to aA & (2) \\ A \to bA & (3) \\ A \to b & (4) \end{array} \right\}$$

Un análisis del tipo de sentencias que pueden derivarse nos lleva fácilmente a la conclusión de que todas terminan con el símbolo b, por aplicación de (4), y todas empiezan por a, pudiendo tener en medio cualquier número de a seguido de cualquier número de b. Obsérvese que para el lenguaje así generado puede darse una expresión regular: $L(G) = aa^* bb^*$.

5. CLASIFICACIÓN DE LAS GRAMÁTICAS Y DE LOS LENGUAJES.

5.1. Gramáticas de tipo 0 ó no restringidas.

La gramática definida de una manera general en el Apartado 1 se llama gramática de tipo 0 ó no restringida. Recordemos que las reglas de escritura o producciones son de la forma $\alpha \rightarrow \beta$, con $\alpha \in E^+$ y $\beta \in E^*$, es decir, la única restricción es que no puede haber reglas de la forma $\Lambda \rightarrow \beta$.

Introduciendo restricciones adicionales en las reglas se obtienen sucesivamente gramáticas cada vez más restringidas. Pasemos a definir y comentar la clasificación debida a Chomsky y aceptada universalmente.

5.2. Gramáticas de tipo 1 o sensibles al contexto (context-sensitive).

En las gramáticas de tipo 1 las reglas son de la forma:

$$\alpha_1 \, A \, \alpha_2 \rightarrow \alpha_1 \, \beta \, \alpha_2$$

con $A \in E_A$; $\alpha_1, \alpha_2 \in E^*$; $\beta \in E^+$

Es decir, A puede reemplazarse por β siempre que esté en el contexto de α_1 y α_2. (Obsérvese que α_1, ó α_2, o ambas, puede ser la cadena vacía, Λ).

Una propiedad importante de las gramáticas de tipo 1 es que *las cadenas que se van obteniendo en cualquier derivación son de longitud no decreciente*. En efecto, al definir las reglas más arriba hemos especificado que $\beta \in E^+$, es decir, $\beta \neq \Lambda$, por lo que $lg(A) = 1 \leq lg(\beta)$, y $lg(\alpha_1 \, A \, \alpha_2) \leq lg(\alpha_1 \, \beta \, \alpha_2)$, o sea, que la longitud del consecuente nunca puede ser menor que la del antecedente. En el siguiente Capítulo demostraremos que

la inversa es también cierta, en el sentido de que, *si todas las reglas de una gramática cumplen la condición de no decrecimiento, se puede hallar una gramática equivalente con reglas sensibles al contexto.*

Salvo en el primero y el segundo, todas las gramáticas definidas en los ejemplos del Apartado 4 son sensibles al contexto. En el ejemplo 1, es la regla 0S1 → 01 la que no cumple la condición, ya que sustituye S por Λ en el contexto (0,1). En cuanto al segundo ejemplo, la regla abA → baab no es del tipo sensible al contexto (lo sería si fuera abA → abab); sin embargo, la longitud del antecedente es menor o igual que la del consecuente en todas las reglas, por lo que habrá una gramática equivalente sensible al contexto. En el siguiente Capítulo veremos cómo se puede encontrar.

5.3. Gramáticas de tipo 2 ó de contexto libre (context-free).

Las gramáticas de tipo 2 son un caso particular de las de tipo 1, con $\alpha_1 = \alpha_2 = \Lambda$, es decir, las reglas son del tipo

$$A \rightarrow \beta$$

con $A \in E_A$, $\beta \in E^+$.

Estas gramáticas, también llamadas gramáticas de Chomsky o C-gramáticas juegan un papel muy importante tanto en la lingüística como en la teoría de lenguajes de programación, por lo que ya antes de la clasificación propuesta por Chomsky fueron descubiertas por diversos autores a partir de puntos de vista bastante diferentes.

Ejemplos de gramáticas de contexto libre son el 3, el 4 y el 5 del Apartado 4.

5.4. Gramáticas de tipo 3 ó regulares.

Las gramáticas de tipo 3, también llamadas gramáticas de Kleene o K-gramáticas son un caso particular de las gramáticas de tipo 2, con reglas de la forma

$$A \to aB \qquad o \qquad A \to a$$

con $A, B \in E_A$; $a \in E_T$

Un ejemplo de K-gramática es el ejemplo 5 del Apartado 4. Obsérvese que en ese ejemplo podíamos representar el lenguaje mediante una expresión regular; esto no es casualidad: como veremos en el Capítulo 4, *los lenguajes generados por las gramáticas de tipo 3 son exactamente los lenguajes regulares que estudiábamos en el Tema "Autómatas"* (y de ahí que a las K-gramáticas también se les llame gramáticas regulares).

6. Jerarquía de Lenguajes.

Llamaremos lenguaje de tipo 0 al generado por una gramática de tipo 0, lenguaje de tipo 1 ó sensible al contexto al generado por una gramática de tipo 1, lenguaje de tipo 2 ó de contexto libre o C-lenguaje al generado por una gramática de tipo 2, y lenguaje de tipo 3 o regular o K-lenguaje, o también, lenguaje de estados finitos al generado por una gramática de tipo 3.

Según se han definido las gramáticas, es evidente que toda gramática regular es de contexto libre, toda gramática de contexto libre es sensible al contexto, y toda gramática sensible al contexto es de tipo 0. Por consiguiente, si llamamos $\{L(G3)\}$, $\{L(G2)\}$, $\{L(G1)\}$ y $\{L(G0)\}$ a los conjuntos de lenguajes de cada tipo, tendremos que:

$$\{L(G3)\} \subset \{L(G2)\} \subset \{L(G1)\} \subset \{L(G0)\} \subset P(E^*)$$

7. LENGUAJES CON LA CADENA VACÍA.

Es fácil constatar que, tal como se han definido las gramáticas, la cadena vacía, Λ, no puede figurar en ningún lenguaje de tipo 1, 2 ó 3. Una gramática es, esencialmente, una expresión en un metalenguaje que permite dar una descripción finita de ciertos lenguajes (no de todos) definidos sobre E. Es obvio que si L tiene una descripción finita, $L_1 = L \cup \{\Lambda\}$ también puede tenerla: bastará añadir de algún modo "Λ también está en L_1" a la descripción de L, y esto puede hacerse agregando $S \to \Lambda$ a las reglas de la gramática que describe a L.

Ahora bien, si habíamos impuesto a las reglas de las gramáticas de tipo 1, $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, la condición de que $\beta \neq \Lambda$ era para conseguir la importante propiedad de no decrecimiento. Si ahora añadimos $S \to \Lambda$, será preciso que S no aparezca en el consecuente de ninguna regla si queremos que se conserve tal propiedad. A este respecto, es importante el siguiente teorema:

Teorema 7.1. Si G es una gramática de tipo 1, 2 ó 3, puede encontrarse otra gramática equivalente, G_1, de tipo 1, 2 ó 3 respectivamente, tal que $L(G_1) = L(G)$, y tal que su símbolo inicial, S_1, no aparece en el consecuente de ninguna regla de G_1. Si $G = \langle E_T, E_A, P, S\rangle$,

$$G_1 = \langle E_T, E_A \cup \{S_1\}, P_1, S_1\rangle, \quad \text{donde}$$

$$P_1 = P \cup \{S_1 \to \alpha \mid (S \to \alpha) \in P\}$$

Demostración:

a) Supongamos que $S \overset{*}{\underset{G}{\Rightarrow}} x$, y sea $S \to \alpha$ la primera regla

utilizada en esa derivación; entonces, $S \underset{G}{\Rightarrow} \alpha \overset{*}{\underset{G}{\Rightarrow}} x$. Por la - definición de P_1, $(S_1 \to \alpha) \in P_1$, de modo que $S_1 \underset{G_1}{\Rightarrow} \alpha$, y como $P \subset P_1$, $\alpha \overset{*}{\underset{G_1}{\Rightarrow}} x$. Por consiguiente, $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} x$, y $L(G) \subset L(G_1)$.

b) Supongamos que $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} y$, siendo $S_1 \to \beta$ la primera - regla utilizada en G_1:

$$S_1 \underset{G_1}{\Rightarrow} \beta \overset{*}{\underset{G_1}{\Rightarrow}} y$$

Si $S_1 \to \beta$ es una regla en G_1, en G deberá existir la regla $S \to \beta$, por lo que $S \underset{G}{\Rightarrow} \beta$. Por otra parte, P_1 se ha definido de modo que S_1 no aparezca en el consecuente de ninguna regla, por lo que no estará incluido en α ni aparecerá en ninguna de las formas sentenciales de la derivación $\beta \overset{*}{\underset{G_1}{\Rightarrow}} y$; entonces, esta derivación será también válida en G: $\beta \overset{*}{\underset{G}{\Rightarrow}} y$. Vemos así que $S \overset{*}{\underset{G}{\Rightarrow}} \beta$, y por tanto, $L(G_1) \subset L(G)$. Teniendo en cuenta el resultado anterior podemos afirmar que $L(G) = L(G_1)$.

c) Tal como se ha definido P_1, es inmediato comprobar que si G es una gramática de tipo 1, 2 ó 3, G_1 es de tipo 1, 2 ó 3 respectivamente.

En virtud este teorema, dada una gramática cualquiera de tipo 1, 2 ó 3, G, podemos pasar a G_1, y de ésta a G_2 añadiendo la regla $S_1 \to \Lambda$, con lo que tendremos $L(G_2) = L(G) \cup \{\Lambda\}$, con G_2 del mismo tipo que G y de longitud no decreciente.

Ejemplo.

Sea G, definida por $E_A = \{S\}$; $E_T = \{0,1\}$; $P = \{(S \to 0S1); (S \to 01)\}$. El lenguaje es $L(G) = \{0^n 1^n \mid n \geq 1\}$. Podemos construir G_1, con $E_{A_1} = \{S, S_1\}$; $E_{T_1} = E_T$; $P_1 = \{(S_1 \to 0S1); (S_1 \to 01);$

$(S \to 0S1);\ (S \to 01)\}$, siendo ahora S_1 el símbolo inicial; es fácil comprobar que $L(G_1) = L(G)$. Entonces, G_2 tendrá $E_{A_2} = E_{A_1} = \{S, S_1\}$; $E_{T_2} = E_T = \{0,1\}$; $P_2 = \{(S_1 \to 0S1);\ (S_1 \to 01);\ (S \to 0S1);\ (S \to 01);\ (S_1 \to \Lambda)\}$, con lo que $L(G_2) = L(G) \cup \{\Lambda\} = \{0^n\ 1^n \mid n \geq 0\}$.

De una manera general, es fácil deducir del Teorema 7.1. el siguiente

Corolario. Si L es un lenguaje de tipo 1, 2 ó 3, entonces $L \cup \{\Lambda\}$ y $L - \{\Lambda\}$ son lenguajes de tipo 1, 2 ó 3 respectivamente.

8. RESUMEN.

En este Capítulo hemos definido los conceptos de gramática, lenguaje generado por una gramática y gramáticas equivalentes. Hemos visto la clasificación de las gramáticas según las restricciones impuestas a sus producciones y cómo esta clasificación da lugar a una jerarquía de lenguajes. Finalmente, hemos estudiado la manera de modificar una gramática para que el lenguaje contenga la cadena vacía sin cambiar de tipo.

9. NOTAS HISTÓRICA Y BIBLIOGRÁFICA.

Los primeros estudios sobre gramáticas formales y la clasificación de las mismas son obra de CHOMSKY (1956), que partía de trabajos previos poco formalizados de lingüistas de la escuela estructuralista americana. Sin duda, Noam Chomsky es la figura más destacada de la lingüística moderna, tanto por desarrollar los fundamentos matemáticos (CHOMSKY y MILLER,

1958; CHOMSKY, 1959), que han sido de gran utilidad en Informática para la descripción de lenguajes artificiales, como por sus teorías sobre el origen y la naturaleza de los lenguajes naturales (CHOMSKY, 1968, 1975), aunque éstas hayan sido fuertemente criticadas (Dos obras del principal crítico de Chomsky están traducidas al español: LURIA (1974 a,b)).

Tres libros recomendables sobre los lenguajes formales en general son el de HARRISON (1978), que, además de reciente, es muy completo y riguroso, el de HOPCROFT y ULLMAN (1969), muy citado y utilizado en muchas universidades, y el de GROSS y LENTIN (1967), que está traducido al español.

10. Ejercicios.

1. Muchas veces interesa que los árboles de derivación en una gramática sean binarios, es decir, que de cada nodo sólo puedan salir uno o dos arcos. ¿Cómo deberán ser las reglas de escritura para que esto sea posible?. Modificar la gramática del ejemplo del Capítulo 1 para que sus árboles sean binarios.

2. Dada la gramática $E_A = \{S,A\}$; $E_T = \{0,1\}$;

$P = \{(S \rightarrow 0A);\ (A \rightarrow 0A);\ (A \rightarrow 1S);\ (A \rightarrow 0)\}$,

a) ¿de qué tipo es?;
b) expresar de algún modo el lenguaje que genera;
c) hallar otra gramática que genere el mismo lenguaje más la cadena vacía.

3. Dar una gramática que permita generar todos los números racionales escritos en decimal con el formato: (signo) (parte entera). (parte fraccionaria).

4. Modificar las gramáticas de los ejemplos del Aparta do 4 para que se obtengan los mismos lenguajes con la cadena - vacía.

CAPITULO 3.

DE ALGUNAS PROPIEDADES DE LOS LENGUAJES

1. INTRODUCCIÓN.

En el estudio de los lenguajes se plantean problemas - que tienen o no solución dependiendo del tipo de gramática. - Por ejemplo, para una gramática, G, de reglas sensibles al contexto, el problema de saber si L(G) es vacío, finito o infinito es, en general, indecidible, mientras que para una gramática de contexto libre es decidible, es decir, existe un algoritmo que, aplicado a G, produce una de las tres respuestas; naturalmente, este problema será indecidible para las gramáticas de tipo 0 y decidible para las regulares. Otro ejemplo es el de - averiguar si dos gramáticas son equivalentes,problema sólo decidible para las gramáticas regulares.

Al aumentar el grado de generalidad de la gramática, - es decir, al ir desde el tipo 3 hacia el tipo 0, el estudio se hace más complejo y aumenta el número de problemas indecidibles.

Desde el punto de vista de aplicación práctica tanto a los lenguajes naturales como artificiales son particularmente importantes las gramáticas de contexto libre. Este Capítulo se dedica, esencialmente, a dos características asociadas a tales gramáticas: la *recursividad* y la *ambigüedad*.

La recursividad es una propiedad válida no sólo para - las gramáticas de contexto libre, sino también para las sensibles al contexto, ya que es una consecuencia del no decreci- - miento de las cadenas. La ambigüedad sólo puede definirse y es

tudiarse cómodamente para las gramáticas de tipo 2 (y 3), en - las que pueden describirse las derivaciones mediante árboles.

En aras de la brevedad omitiremos la demostración de - algunos teoremas.

2. No decrecimiento en las gramáticas sensibles al contexto.

En el Capítulo anterior vimos que como consecuencia de la misma definición de reglas sensibles al contexto ($\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2, \beta \neq \Lambda$) se desprende inmediatamente que la longitud de las cadenas derivadas por aplicación de tales reglas no puede disminuir. Nos proponemos ahora demostrar la inversa, que ya - no es tan evidente:

Teorema 2.1. Dada una gramática G_1 con reglas de longitud no decreciente, puede encontrarse otra gramática G_2 equivalente a G_1 cuyas reglas son sensibles al contexto.

Demostración.

Sea $\alpha \rightarrow \beta$ una regla de G_1, y sea $\alpha = p_1 p_2 \ldots p_\ell$ y $\beta = q_1 q_2 \ldots q_{\ell+m}$ con $p_1, q_j \in E$ y $m \geq 0$. Sustituiremos esta regla por un conjunto de reglas sensibles al contexto tales que juntas producen la derivación $\alpha \overset{*}{\underset{G_2}{\Rightarrow}} \beta$ sin aumentar la capacidad generativa.

Para ello ampliamos el alfabeto auxiliar de G_1 con un símbolo R y le añadimos también un símbolo P_1 por cada $p_1 \in E_T$ -

Sustituimos en principio

$$p_1 p_2 p_3 \ldots\ldots p_\ell \rightarrow q_1 q_2 q_3 \ldots\ldots q_{\ell+m}$$

$$p_1p_2p_3\cdots p_{\ell-1}p_\ell \rightarrow Rp_2p_3\cdots p_{\ell-1}p_\ell$$

$$Rp_2p_3\cdots p_{\ell-1}p_\ell \rightarrow Rq_2p_3\cdots p_{\ell-1}p_\ell$$

$$Rq_2p_3\cdots p_{\ell-1}p_\ell \rightarrow Rq_2q_3\cdots p_{\ell-1}p_\ell$$

$$\cdots\cdots\cdots\cdots \qquad \cdots\cdots\cdots\cdots$$

$$Rq_2q_3\cdots q_{\ell-1}p_\ell \rightarrow Rq_2q_3\cdots q_{\ell-1}\cdots q_{\ell+m}$$

$$Rq_2q_3\cdots q_{\ell+m} \rightarrow q_1q_2\cdots q_{\ell+m}$$

Es evidente que:

a) Con este conjunto de reglas se obtiene la derivación

$$p_1p_2\cdots p_\ell \overset{*}{\Rightarrow} q_1q_2\cdots q_{\ell+m}$$

b) Al ser R un símbolo auxiliar no pueden producirse más sentencias que las que pueda producir $\alpha \rightarrow \beta$.

c) Las reglas introducidas sustituyen un p_i por un símbolo o una cadena en un contexto. Por tanto, son del tipo sensible al contexto, salvo si $p_i \in E_T$. En este caso, sustituimos p_i por $P_i \in E_A$ e introducimos la regla terminal $P_i \rightarrow p_i$.

Ejemplo:

Consideremos el ejemplo 2 del anterior Capítulo. Sustituiremos la regla abA → baab por:

abA → RbA
RbA → RaA
RaA → Raab
Raab → baab

Sustituimos a y b por P_1 y P_2 e introducimos las correspondientes reglas terminales, con lo cual obtenemos la gramática equivalente con reglas sensibles al contexto:

$$E_T = \{a,b\};\ E_A = \{S,\ A,\ R,\ P_1\ P_2\};$$

$$P = \left\{ \begin{array}{llll} S & \to P_1P_2AS & S & \to P_1 \\ P_1P_2A & \to RP_2A & A & \to P_2 \\ RP_2A & \to RP_1A & P_1 & \to a \\ RP_1A & \to RP_1P_1P_2 & P_2 & \to b \\ RP_1P_1P_2 & \to P_2P_1P_1P_2 & & \end{array} \right\}$$

3. RECURSIVIDAD DE LOS LENGUAJES SENSIBLES AL CONTEXTO.

Una gramática es un algoritmo para generar un lenguaje, pero un problema de gran importancia es el del reconocimiento, es decir, dados un lenguaje $L \subset E^*$ y una cadena $x \in E^*$, ¿existe un algoritmo para determinar si $x \in L$ o $x \notin L$?. Si, por ejemplo, L es un lenguaje de programación y x un programa, es importante poder determinar si el programa es correcto ($x \in L$) o no ($x \notin L$). El problema del reconocimiento está intimamente ligado al concepto de recursividad (ver Tema "Algoritmos", Cap. 5). En efecto, decir que un lenguaje $L \subset E^*$ es recursivo es lo mismo que decir que existe un algoritmo para calcular la función característica de todo $x \in E^*$, o, lo que es lo mismo, para determinar si $x \in L$ o $x \notin L$.

Teorema 3.1. Todo lenguaje generado por una gramática de reglas sensibles al contexto es recursivo.

Aunque no daremos una demostración de este importante teorema, puede fácilmente intuirse que es una consecuencia de la propiedad de no decrecimiento. En efecto, un lenguaje de tipo 0 es recursivamente numerable, ya que existe un algoritmo -(la gramática de tipo 0) para generar sus elementos (senten--

cias). Dado un elemento $x \in L$, podemos compararlo con cada uno de los elementos generados hasta comprobar que coinciden, pero si $x \notin L$ habría que generar las infinitas sentencias para ver que efectivamente $x \notin L$; es decir, en general, un lenguaje de tipo 0 no es recursivo. Sin embargo, si el lenguaje cumple la propiedad de no decrecimiento (tipo 1 en adelante), tenemos un algoritmo para generar primero todas las sentencias de longitud 1, luego las de longitud 2, etc.; si $lg(x) = \ell$, generaremos todas las sentencias de longitud ℓ, y si x no se encuentra entre ellas, entonces $x \notin L$.

Podemos preguntarnos si la inversa es también cierta, es decir, si todo lenguaje recursivo puede ser generado por una gramática sensible al contexto. La contestación es "no":

Teorema 3.2. Existen lenguajes recursivos que no son sensibles al contexto.

4. ARBOLES DE DERIVACIÓN PARA LAS GRAMÁTICAS DE CONTEXTO LIBRE.

En el Capítulo 1 ya hemos utilizado árboles como un método gráfico para ilustrar las derivaciones en ciertas gramáticas.

Definición 4.1. Un árbol es un conjunto finito de nodos unidos por arcos orientados (diremos que un arco sale del nodo n_i y entra en el nodo n_j), cumpliéndose tres condiciones:

1. Existe un nodo, y sólo uno, llamado raíz, en el que no entra ningún arco.

2. Para todo nodo n_m existe una secuencia de arcos, y sólo una, tal que el primero sale de la raíz y entra en n_i, el segundo sale de n_i y entra en n_{i+1}, -

etc., y el último entra en n_m.

3. En cada nodo (salvo la raíz) entra un arco y sólo uno.

Un nodo n_i es descendiente directo de otro nodo n_j si existe un arco que sale de n_j y entra en n_i. Un nodo n_m es descendiente de otro n_o si existe una secuencia n_1, n_2....n_{m-1} tal que n_m es descendiente directo de n_{m-1}, n_{m-1} lo es de n_{m-2},.... n_1 lo es de n_o.

Se llaman hojas a los nodos que no tienen ningún descendiente.

Entre los nodos de un árbol se puede establecer una *relación de orden*: todos los descendientes directos de n se pueden ordenar de *izquierda a derecha*, y si n_1 está a la izquierda de n_2 todos los descendientes de n_1 estarán a la izquierda de n_2.

Definición 4.2. Dada una gramática de contexto libre $G = <E_A, E_T, P, S>$, un árbol es un árbol de derivación en G si:

1. Cada nodo tiene una etiqueta que es un símbolo de $E = E_A \cup E_T$: $Eti(n) \in E$
2. $Eti(raíz) = S$
3. Si n no es una hoja, $Eti(n) \in E_A$.
4. Si $n_1, n_2 \ldots . n_k$ son todos los descendientes directos de n de izquierda a derecha y $Eti(n) = A$, $Eti(n_1) = \ell_1$, $Eti(n_2) = \ell_2$,....$Eti(n_k) = \ell_k$, entonces $(A \rightarrow \ell_1 \ell_2 \ldots . \ell_k) \in P$.

Llamaremos resultado de un árbol de derivación a la cadena compuesta por las etiquetas de las hojas leídas de izquier

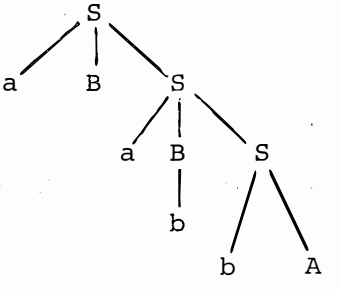
da a derecha.

Ejemplo 4.3.

$$E_A = \{S, A, B\} \; ; \quad E_T = \{a,b\}$$

$$P = \left\{ \begin{array}{llll} S \to aB & (1) & A \to a & (5) \\ S \to bA & (2) & A \to bAA & (6) \\ S \to aBS & (3) & B \to b & (7) \\ S \to bAS & (4) & B \to aBB & (8) \end{array} \right\}$$

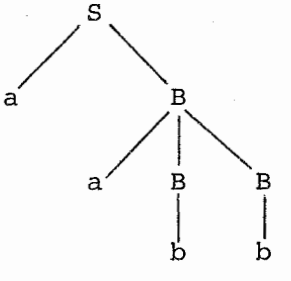
El árbol



Tiene como resultado la cadena aBabbA, y corresponde a la derivación

$$S \overset{}{\Rightarrow}_{3} aBS \overset{}{\Rightarrow}_{3} aBaBS \overset{}{\Rightarrow}_{7} aBabS \overset{}{\Rightarrow}_{2} aBabbA$$

El árbol



tiene como resultado la sentencia aabb, y corresponde a la derivación

$$S \overset{}{\Rightarrow}_{1} aB \overset{}{\Rightarrow}_{8} aaBB \overset{}{\Rightarrow}_{7} aabB \overset{}{\Rightarrow}_{7} aabb$$

Teorema 4.4. Dada una gramática de contexto libre, G, $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ si y sólo si existe un árbol de derivación en G con resultado α.

5. AMBIGÜEDAD EN LAS GRAMÁTICAS DE CONTEXTO LIBRE.

Definición 5.1. Una gramática de contexto libre G se dice que es ambigua si existe al menos una sentencia en L(G) que puede obtenerse por dos o más derivaciones distintas, o lo que es lo mismo, le corresponden dos o más árboles diferentes. El número de árboles caracteriza el grado de ambigüedad de la sentencia.

Ejemplo 5.2.

Consideremos el siguiente subconjunto de reglas del castellano:

<Sentencia> → <SN> <SP>

<SN> → <Det> <Ncom> <Sprep>

<SN> → <Det> <Ncom>

<SP> → <Vintr> <Sprep>

<SP> → <Sprep> <Vintr> <Sprep>

<Sprep> → <Prep> <SN>

<Det> → <Art>

<Art> → los

<Art> → el

<Art> → la

<Ncom> → partidos

<Ncom> → TV

<Ncom> → personal

<Vintr> → aburren

<Prep> → de

<Prep> → a

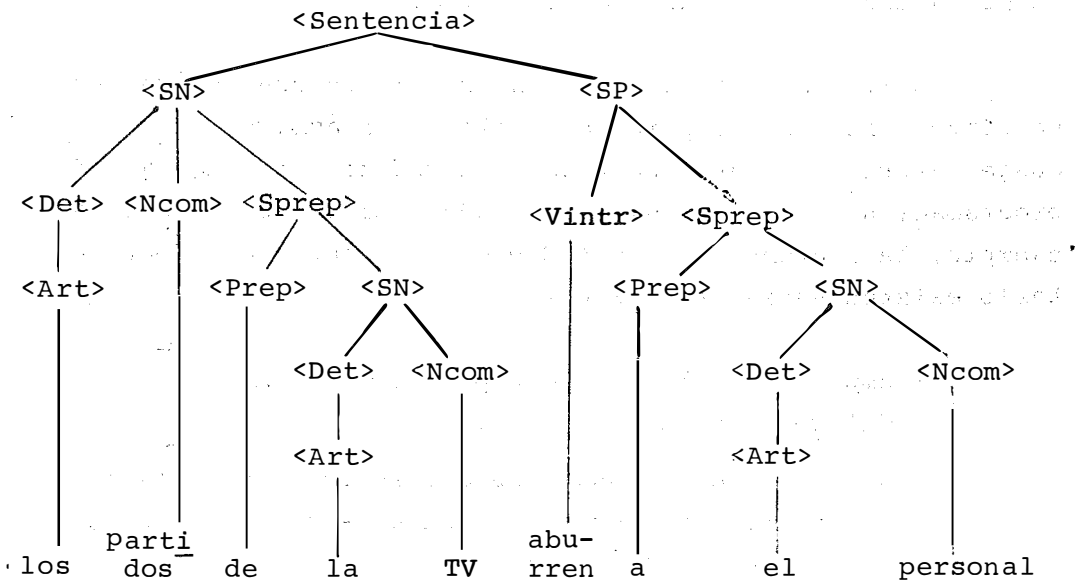
La sentencia

Los partidos de la TV
aburren al personal

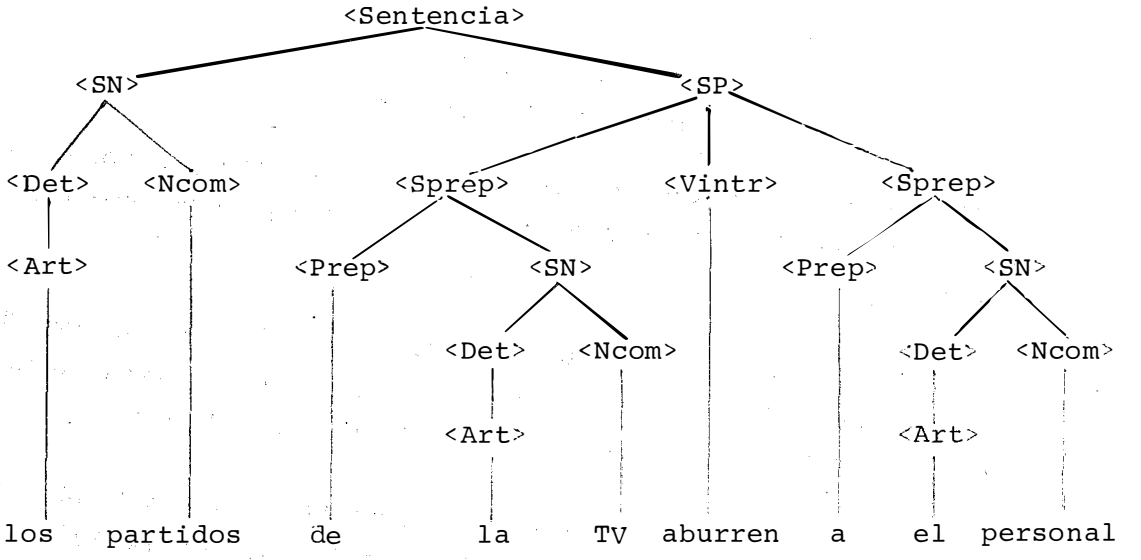
es una sentencia ambigua en la gramática definida por tales reglas (que, por tanto, es una gramática ambigua, como enseguida veremos.

De momento, existe una ambigüedad producida por los diversos significados de la palabra "partidos", pero ésta es una ambigüedad semántica, que no es la que aquí nos ocupa.

Según la definición 5.1, la sentencia será ambigua si puede ser derivada de varias formas (es lo que los linguistas llaman "ambigüedad extructural"). Y, en efecto, a la sentencia en cuestión puede corresponderle este árbol de derivación:



o también este otro:



El primero corresponde al sentido inmediato de la sentencia: "los partidos *de la TV.....*" o "los partidos *que se programan en la TV...*", mientras que el segundo indica una construcción alternativa y no muy feliz de "los partidos aburren *al personal de la TV*", que podría haber sido ingeniada por un poeta mediocre en busca de una rima.

Así, los lenguajes de contexto libre son interesantes en lingüística porque permiten reflejar la ambigüedad del lenguaje natural. En las gramáticas generadoras de lenguajes de programación es preciso eliminar toda posible ambigüedad. Por ejemplo, la sentencia A + B * C puede ser ambigua, y para evitarlo existen varias soluciones:

- ampliar el alfabeto con paréntesis: (A + B) * C es diferente de A + (B * C);
- dar prioridad a unos operadores sobre otros: si * tiene prioridad sobre +, entonces A + B * C es lo mismo que A + (B * C);

- utilizar la llamada *notación polaca* que explicaremos en el Capítulo 5 y que permite prescindir de los paréntesis: A + (B * C) se escribirá ABC*+, mientras que (A + B) * C se escribirá AB + C*.

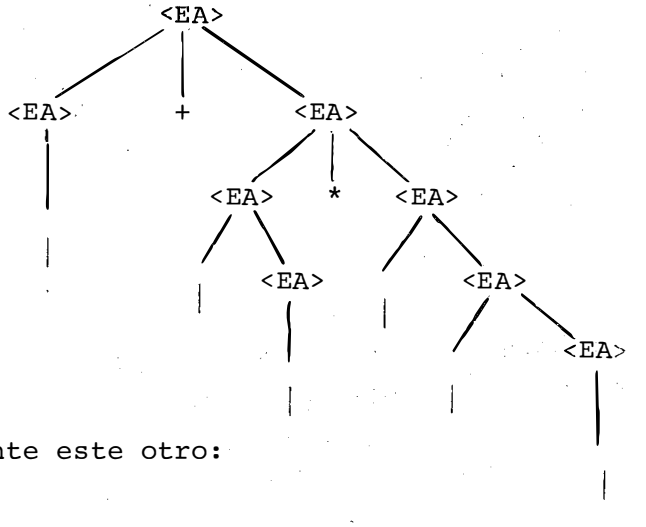
Ejemplo 5.3.

E_A = {<EA>}; E_T ={|, +, *}; S = <EA>

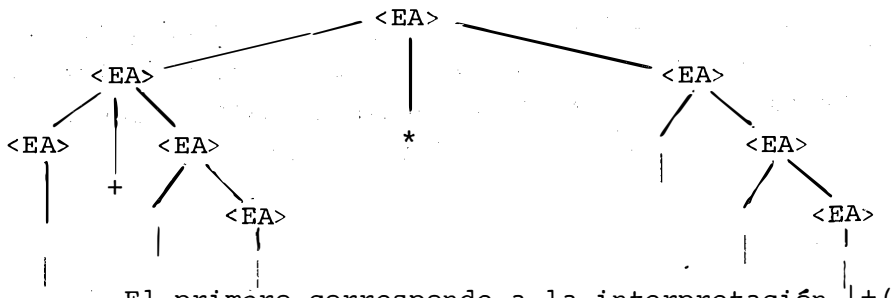
("EA" significa "expresión aritmética")

$$P = \left\{ \begin{array}{l} \text{<EA>} \rightarrow \text{<EA>} + \text{<EA>} \\ \text{<EA>} \rightarrow \text{<EA>} * \text{<EA>} \\ \text{<EA>} \rightarrow \text{|<EA>} \\ \text{<EA>} \rightarrow \text{|} \end{array} \right\}$$

Si admitimos que una sucesión de n |'s representa el número natural n, esta gramática genera sentencias que representan combinaciones de los números naturales con las operaciones de suma y producto. Consideremos la sentencia |+||*|||; la derivación puede hacerse mediante el árbol:



o también mediante este otro:

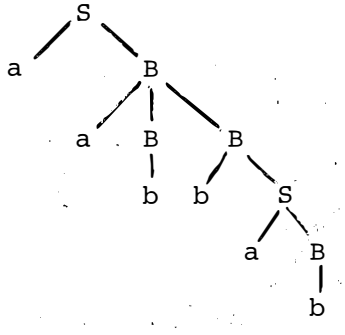


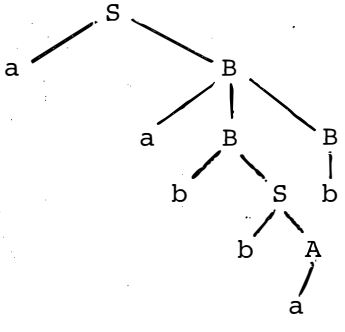
El primero corresponde a la interpretación |+(||*|||) (prioridad de *), es decir, 7, mientras que el segundo corresponde a (| + ||) * ||| (prioridad de +), es decir 9.

Puede ocurrir que un mismo lenguaje pueda ser generado por una gramática ambigua y por otra gramática no ambigua. Un lenguaje es inherentemente ambiguo si todas las gramáticas que lo generan son ambiguas.

Ejemplo 5.4.

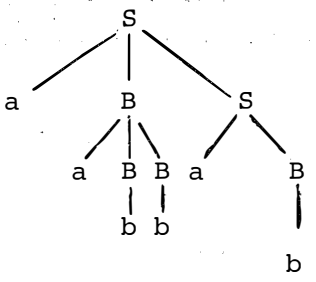
La gramática del ejemplo 3 del Capítulo 2 es ambigua. En efecto, la sentencia aabbab, por ejemplo, puede derivarse según indican los siguientes árboles:





Sin embargo, la gramática del ejemplo 4.3 de este Capítulo es equivalente a ella y no ambigua (HOPCROFT y ULLMAN, 1969), por lo que el lenguaje generado (conjunto de cadenas que tienen igual número de a's que de b's) no es inherentemente ambiguo. La derivación de la misma cadena en esta segunda

gramática se representará por el árbol:



El problema de la ambigüedad, desde un punto de vista general, es indecidible:

Teorema 5.5. No existe un algoritmo que, aplicado a una C-gramática arbitraria, permita decidir si la gramática es ambigua o no.

6. La notación de Backus y un ejemplo.

En la notación de Backus, también llamada BNF (*Backus Normal Form o Backus-Naur Form*), en lugar de utilizarse el símbolo "→" se utiliza "::=", y, además, se pueden abreviar varias reglas del tipo $A::=\alpha_1$, $A::=\alpha_2$,...,$A::\alpha_n$ en una sola: $A:: = \alpha_1|\alpha_2|...|\alpha_n$.

Nosotros seguiremos utilizando el símbolo "→", aunque aprovecharemos la posibilidad de abreviar las reglas.

Como ejemplo para ilustrar la forma en que puede construirse una gramática de contexto libre para un lenguaje de programación vamos a dar algunas reglas de una posible gramática para ENSAM. Creemos que el lector no encontrará especiales dificultades para desarrollar la gramática completa.

Tomaremos como alfabeto terminal el conjunto de todos los caracteres (letras, cifras y caracteres especiales), que

completaremos con b (blanco, o espacio) y cl (cambio de línea o de ficha). El alfabeto auxiliar será: <Programa>, <Instrucción>, etc. (Todas las categorías sintácticas que necesitemos). El símbolo inicial será <Programa>.

He aquí algunas reglas:

<Programa> → <Línea> cl <Programa> | FIN
<Línea> → <Instrucción> | <Pseudo instr.> | <Comentario>
<Instrucción> → <Eti> <CO> <I> <Dir> <CD>
<ETI> → b^9 | <Letra> <Resto>
<Resto> → b^8 | <Carácter> b^7 | <Carácter> <Carácter> b^6 | ...
..... | <Carácter>^{5} b^3
<CO> → SUM|RES|.....
<I> → b^7 | * b^6
<Dir> → <Eti> | <NO-16383>
<CD> → b|, <RI>|,<RI><Inc> |,,<Inc>
<RI> → 0|1|2|3
<Inc> → <NO-16383>
<Pseudo> → <Eti> <COP>
<COP> →REM b^7 <NO-16393> | CEN b^7 <CEN> .....
etc.

Es fácil comprobar que existen muy diversas posibilidades para definir las reglas, lo que conduce a tantas otras gramáticas equivalentes.

7. RESUMEN.

Los lenguajes de tipo 0 son recursivamente numerables pero no recursivos. Los lenguajes de tipo 1 (y 2 y 3) son recursivos: existe un algoritmo para decidir si $x \in L$ o $x \notin L$. Sin embargo, hay lenguajes recursivos que no son de tipo 1. Es decir:

$$\{L(G3)\} \subset \{L(G2)\} \subset \{L(G1\} \subset \{L_{\text{recursivos}}\} \subset \{L(G0)\} =$$

$$= \{L_{\text{recursivamente num}}\}$$

En los lenguajes de contexto libre las derivaciones se pueden representar gráficamente mediante árboles de derivación. A cada derivación corresponde un árbol y viceversa. Si alguna sentencia puede derivarse de dos o más formas diferentes se dice que la gramática es ambigua.

8. NOTAS HISTÓRICA Y BIBLIOGRÁFICA.

La mayor parte de los resultados concernientes a los lenguajes sensibles al contexto y, especialmente, a los de contexto libre, así como sus aplicaciones tanto a lenguajes naturales como a lenguajes de programación aparecieron durante los años 60. Las primeras propiedades indecidibles de las C-gramáticas se publicaron por BAR-HILLEL, PERLES y SHAMIR (1961), y la indecidibilidad de la ambigüedad se demostró por CANTOR - - (1962), FLOYD (1962) y CHOMSKY y SCHUTZENBERGER (1963), inde-- pendientemente. FLOYD (1964) publicó una revisión sobre la - - aplicación de las gramáticas formales a los lenguajes de pro-- gramación.

La bibliografía, especialmente sobre C-gramáticas, es muy amplia; como botón de muestra podemos citar el libro de - GINSBURG (1966), del cual se puede encontrar un resumen en - - GINSBURG (1968).

Hay gran cantidad de resultados sobre propiedades indecidibles de las C-gramáticas y sobre simplificación de las mismas o reducción a unas llamadas "formas normales", que pueden estudiarse en libros específicos, como el citado más arriba, o de carácter general, como los tres recomendados en el Capítulo

anterior.

La notación de Backus fue introducida (BACKUS, 1959) - para describir la gramática del "International Algebraic Len-- guaje", origen del ALGOL 60 (NAUR, 1963).

9. EJERCICIOS.

1. Dada la gramática definida por

$$E_A = \{S\};\ E_T = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$P = \left\{ \begin{array}{l} S \to 12S \\ 12 \to 345 \\ 45 \to 678 \\ S \to 9 \end{array} \right\}$$

construir una gramática equivalente con reglas sensibles al - contexto.

2. Dada la gramática definida por

$$E_A = \{S, A, B\};\quad E_T = \{0, 1\};$$

$$P = \left\{ \begin{array}{ll} S \to 0A & B \to 1S0 \\ S \to 1B & B \to 0 \\ A \to 0S & \end{array} \right\}$$

determinar, de las siguientes sentencias, cuáles pertenecen y cuáles no al lenguaje: 00; 10; 100; 1100; 1010.

3. Hallar una gramática de contexto libre que genere las sentencias aritméticas de FORTRAN.

4. ¿De qué tipo es una gramática cuyas reglas son todas de la forma $A \to x\ B$ ó $A \to x$, con $A, B \in E_A$, $x \in E_T^*$?; de-

mostrar que siempre puede encontrarse una gramática regular - equivalente.

5. Con $E_A = \{S\}$; $E_T = \{a, b, c\}$, la gramática G_1 con reglas

$$P_1 = \left\{ \begin{array}{l} S \rightarrow aSa \\ S \rightarrow bSb \\ S \rightarrow aaSaa \\ S \rightarrow c \end{array} \right\}$$

es una gramática ambigua, mientras que G_2, con

$$P_2 = \left\{ \begin{array}{l} S \rightarrow aSa \\ S \rightarrow bSb \\ S \rightarrow c \end{array} \right\}$$

es equivalente a G_1 y no es ambigua. Comprobarlo haciendo algunas derivaciones. Describir informalmente la forma general de las sentencias del lenguaje.

CAPITULO 4.

LENGUAJES Y AUTOMATAS

1. INTRODUCCIÓN.

En este Capítulo vamos a plantear el problema del reconocimiento de lenguajes desde un punto de vista material, ya que estudiaremos las relaciones entre el tipo de lenguaje y la estructura de máquina capaz de reconocerlo; estas relaciones pueden enfocarse en dos sentidos:

a) dada una gramática, G, ¿qué estructura deberá tener una máquina, M, tal que L(M) = L(G)? (es decir, lenguaje reconocido por M igual a lenguaje generado por G);

b) dada una máquina, M, ¿Cuál será la gramática, G, tal que L(G) = L(M)?.

De anteriores Temas conocemos dos estructuras de máquina que pueden ser utilizadas como reconocedores de cadenas de símbolos: el autómata finito (Tema "Autómatas") y la máquina de Turing (Tema "Algoritmos"). Veremos aquí que la clase de lenguajes que pueden ser reconocidos por autómatas finitos es precisamente la clase de lenguajes que pueden ser generados por una gramática de tipo 3 (regular), y análogamente para las MT y las gramáticas de tipo 0. Para los dos tipos intermedios de gramáticas definiremos unos autómatas que pueden considerarse como restricciones de la máquina de Turing o como extensiones del autómata finito. Tendremos así una jerarquía de máqui-

nas paralela a la jerarquía de lenguajes (*).

Reflexione el lector en el hecho de que las "clases de lenguajes" (la clase de los lenguajes generados por una gramática regular, la clase de los lenguajes reconocidos por un autómata finito, etc.) son subconjuntos de $P(E^*)$ (conjunto de las partes de E^*).

El desarrollo completo de las ideas expuestas exige la demostración de una serie de teoremas acompañada de las correspondientes construcciones (dada una gramática de tipo 0, diseñar la correspondientes MT y viceverse, etc.) que alargaría excesivamente este Tema, por lo que nos contentaremos con desarrollar únicamente el caso más sencillo (gramáticas regulares y autómatas finitos), limitándonos a enunciar los teoremas en los otros tres.

2. Lenguajes de tipo 0 y máquinas de Turing.

2.1. Reconocedor de Turing.

En el Tema "Algoritmos" se define la parte de control de una MT como

$$T - Q = \langle E, (E \times M) \cup (Stop), Q, f, g \rangle$$

Al igual que en el Tema "Autómatas" adaptábamos la definición general de autómata finito para especializarlo como reconocedor, adaptemos también la definición de T - Q. Primero, explicaremos de una manera informal lo que entendemos por má--

(*) Obsérvese que un AF se puede considerar como un caso muy particular de MT que sólo puede leer de la cinta, ésta se desplaza en un solo sentido, y no para.

quina de Turing reconocedorea o reconocedor de Turing.

Un reconocedor de Turing será una MT que, inicializada en un estado inicial predeterminado, q_1, y teniendo la cadena $x \in E^*$ en su cinta (descripción instántanea inicial $0q_1x0$) opera de tal modo que después de un tiempo finito se para, tras escribir un 1, si $x \in L$. Si $x \notin L$ pueden ocurrir dos cosas: escribe 0 y se para (rechaza x), o no se para. (Suponemos que 0, 1 $\in$ E).

T - Q es un autómata finito que siempre se podrá describir como una máquina de Moore, con una función de salida:

$$h: Q \to (E \times M) \cup (\text{Stop})$$

Definiremos los estados finales de aceptación como aquellos para los que la función h es "stop", siendo la h del estado anterior "1". Tendremos así un conjunto $F_A \subset Q$ de estados de aceptación:

$$F_A = \{q_A \mid h(q_A) = \text{Stop y } h(q_A,) = 1, \text{ siendo } q_A, \text{ tal que } f(e, q_A,) = q_A \text{ para algún } e \in E\}$$

El control de un reconocedor de Turing será definido como

$$R_{T-Q} = \langle E, Q, f, q_1, F_A \rangle,$$

y el reconocedor será $R_{MT} = R_{T-Q}$ + cinta.

2.2. Lenguaje aceptado por un reconocedor de Turing.

Podemos ya definir el lenguaje aceptado por un R_{MT}:

$$L(R_{MT}) = \{x \in E^* \mid \text{Res } R_{MT} (0q_1x0) = 0yq_Az0;\ q_A \in F_A; y,z \in E^*\}$$

y enunciar los dos teoremas que establecen la *equivalencia entre la clase de lenguajes aceptados por algún* R_{MT} *y la clase de lenguajes generados por una gramática de tipo 0:*

2.3. Teorema MT1.

Para toda gramática de tipo 0, G0, existe un reconocedor de Turing, R_{MT}, tal que $L(R_{MT}) = L(G0)$.

Obsérvese que, en general, no tiene por qué existir R'_{MT} tal que $L(R'_{MT}) = E^* - L(G0)$; esto sólo ocurrirá en el caso de que L(G0) sea recursivo, y en tal caso R'_{MT} puede ser el mismo R_{MT}, definiendo los estados de aceptación por $h(q_A,) = 0$.

2.4. Teorema MT2.

Para todo reconocedor de Turing, R_{MT}, existe una gramática de tipo 0, G0, tal que $L(G0) = L(R_{MT})$.

2.5. Conclusión.

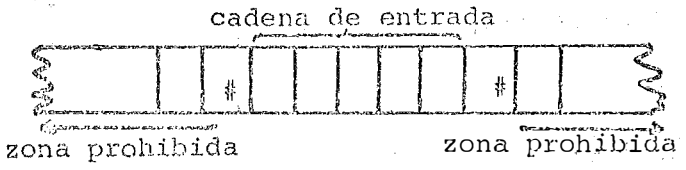
El teorema MT1 viene a decir que $\{L(G0)\} \subset \{L(R_{MT})\}$, y el MT2, que $\{L(R_{MT})\} \subset \{L(G0)\}$. Además, sabemos que la clase de lenguajes reconocibles por un R_{MT} coincide con la clase de los lenguajes recursivamente numerables (esto es una consecuencia de la hipótesis de Turing), y que los lenguajes recursivos son un subconjunto de éstos, luego, como conclusión, escribiremos:

$$\{L_{recurs.}\} \subset \{L_{recurs.\ num.}\} = \{L(R_{MT})\} = \{L(G0)\}$$

3. LENGUAJES SENSIBLES AL CONTEXTO Y AUTÓMATAS LIMITADOS LINEALMENTE.

3.1. Autómata limitado linealmente.

Un autómata limitado linealmente, ALL, es una MT cuya cabeza no puede desplazarse fuera de los límites entre los que se sitúa inicialmente la cadena de entrada. Para señalar tales límites, se incluye en E un símbolo especial como marcador, #:



Un reconocedor limitado linealmente, R_{ALL}, se definirá por estados finales de aceptación con un estado inicial q_1, igual que un R_{MT}.

3.2. Lenguaje aceptado por un reconocedor limitado linealmente.

La definición del lenguaje aceptado por un R_{ALL} es análoga a la de $L(R_{MT})$, salvo que # no puede utilizarse como símbolo en las cadenas de entrada y que se preserva la distancia entre los dos marcadores:

$$L(R_{ALL}) = \{x \in (E - \#)^* \mid \text{Res } R_{ALL}(\# \; q_1 \; x \; \#) = \# \; y q_A z \#$$

$$q_A \in F_A;\; y,\; z \in (E - \#)^*;\; lg(x) = lg(yz)\}$$

3.3. Teorema ALL1.

Para toda gramática sensible al contexto, G1, existe un reconocedor limitado linealmente, R_{ALL}, tal que $L(R_{ALL}) = L(G1)$.

Como las gramáticas de tipo 1 generan lenguajes recursivos, siempre existirá también R'_{ALL} tal que $L(R'_{ALL}) = E^* - L(G1)$.

3.4. Teorema ALL2.

Para todo reconocedor limitado linealmente, R_{ALL}, existe una gramática sensible al contexto, G1, tal que $L(G1) = L(R_{ALL})$.

3.5. Conclusión.

De los dos teoremas anteriores deducimos:

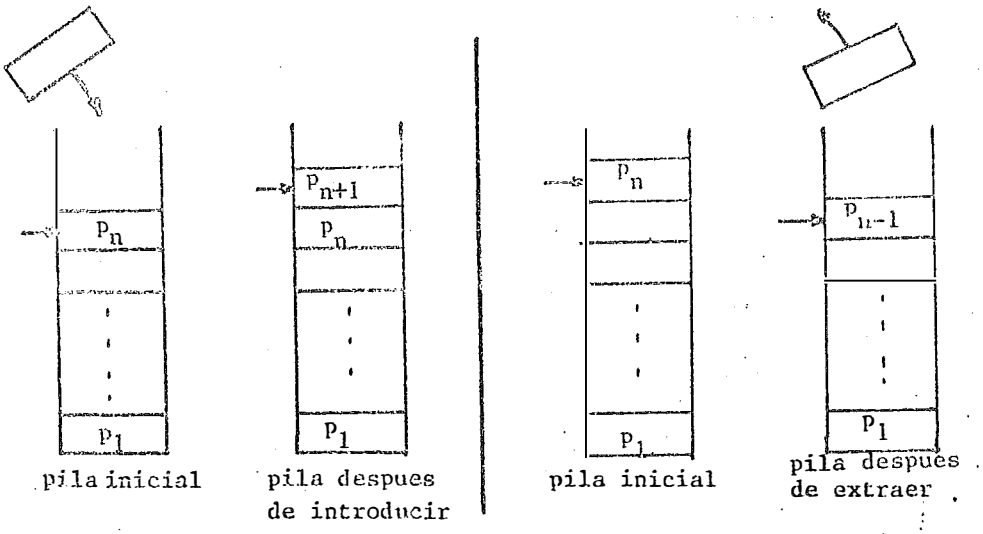
$$\{L(R_{ALL})\} = \{L(G1)\}$$

4. LENGUAJES DE CONTEXTO LIBRE Y AUTÓMATAS DE PILA.

4.1. Autómata de pila.

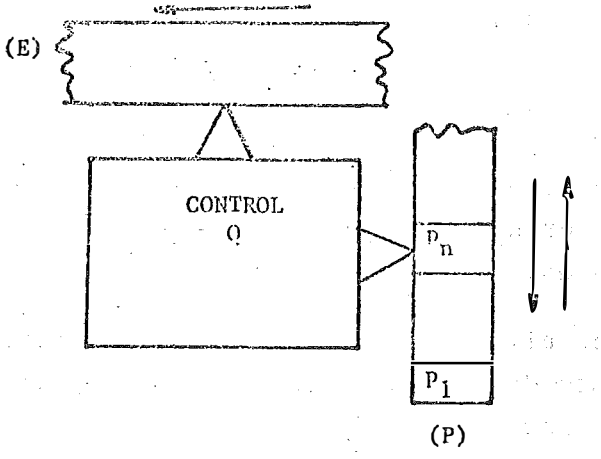
Una pila es un tipo de almacenamiento en el que sólo se pueden leer las informaciones en el orden inverso en que han sido escritas, siguiendo el principio de "*el último en entrar será el primero en salir*", y de ahí que también se le llame memoria LIFO (*last-in, first-out*).

Las dos operaciones posibles con una pila son introducir (escribir) o extraer (leer):



El símbolo " → " en las figuras anteriores indica en - cada momento cuál es la "cabeza" o "cima" de la pila (*).

Un autómata de pila, AP, es un autómata finito al que se le añade una pila potencialmente infinita para guardar re-- sultados intermedios. Puede considerársele como una MT con dos cintas y dos cabezas:



(*) Si la pila se simula en la memoria central de un ordenador mediante un programa, existirá un puntero a la cima de la pila, es decir, una variable que se actualizará en cada - operación para que su valor sea la dirección de la cima.

La cinta (E) se desplaza en un solo sentido, y la correspondiente cabeza sólo puede leer. En la cinta (P) (limitada en un extremo) puede leerse, en cuyo caso desaparece p_n y se desplaza hacia arriba (según la figura) o escribirse, para lo cual se desplaza hacia abajo y se introduce p_{n+1} sobre p_n.

Las operaciones elementales que puede realizar un AP son de dos tipos:

- dependientes de E: se lee un $e_i \in E$, se desplaza la cinta (E), y, en función de e_i, q_j, p_k, el control pasa a otro estado q_1 y en la pila (P) se introduce p_{k+1}, o se extrae p_k, o no se hace nada.

- independientes de E: lo mismo, sólo que e_i no interviene y (E) no se mueve, lo que permite manejar la pila sin tener en cuenta las informaciones de entrada.

En cualquier caso, si se vacía la pila (es decir, se extrae p_1) el AP se para.

4.2. Lenguaje aceptado por un reconocedor de pila

El reconocedor de pila, R_{AP}, será un AP inicializado en un estado q_1 y con p_1 como única información en la cinta (P). La aceptación de cadenas, y, consecuentemente, el lenguaje aceptado por R_{AP}, $L(R_{AP})$ se puede sefinir según dos criterios diferentes:

- por pila vacía: $x \in L(R_{AP})$ si al leer el último símbolo de x la pila queda vacía (y, por tanto, el AP se para).

- por estados finales: $x \in L(R_{AP})$ si al leer el último símbolo de x el control queda en un estado $q_A \in F_A$, siendo $F_A \subset Q$ un conjunto de estados definidos a priori como estados finales de aceptación.

En un estudio formalizado de los AP, que aquí no abordamos, se demuestra que *ambas definiciones de* $L(R_{AP})$ *son equivalentes*, en el sentido de que la clase de los lenguajes aceptados por las estructuras del tipo AP es la misma en ambos casos.

4.3. Teorema AP1.

Para toda gramática de contexto libre, G2, existe un reconocedor de pila, R_{AP}, tal que $L(R_{AP}) = L(G2)$.

4.4. Teorema AP2.

Para todo reconocedor de pila, R_{AP}, existe una gramática de contexto libre, G2, tal que $L(G2) = L(R_{AP})$.

4.5. Conclusión.

De los teoremas enunciados se desprende que

$$\{L(R_{AP})\} = \{L(G2)\}$$

5. LENGUAJES REGULARES Y AUTÓMATAS FINITOS.

5.1. Autómata finito no determinista.

En el Tema "Autómatas" se han estudiado los reconocedores finitos y se ha visto que la clase de lenguajes que pueden reconocer son los lenguajes regulares. Aquí vamos a demostrar que esa clase de lenguajes coincide precisamente con la clase de lenguajes que pueden ser generados por gramáticas de tipo 3 ó regulares. Para ello tenemos que recurrir a un concepto auxiliar: el AF no determinista. Su necesidad aparece claramente teniendo en cuenta cómo se establecen las relaciones entre gramáticas de tipo 3 y autómatas. En efecto, si en un AF exis-

te una transición del estado A al estado B bajo el efecto de - la entrada e, veremos que la gramática correspondiente contiene la producción A → eB, y a la inversa. Ahora bien, nada impide de que una gramática regular contenga simultáneamente las re-- glas A → eB y A → eC, lo que nos lleva a considerar la posibilidad de que *del estado A con entrada e se pase bien al estado B, bien al C*. Un AF en el que tal cosa puede ocurrir se llama *no determinista*. Quede bien claro que no es la idea de probabilidad la que aquí interviene (por eso ARBIB (1969) sugiere que sería más adecuado el nombre de *posibilista*). El AF no determinista debe ser considerado como un concepto abstracto, desprovisto de interpretación física.

Definición 5.1.1. Un autómata finito no determinista - es una quíntupla

$$AFND = \langle E, S, Q, f, g \rangle,$$

donde todo es igual que en un autómata finito (Tema "Autómatas", Cap. 2, Ap. 1.1), salvo que el rango de la función de transi-- ción no es Q, sino P(Q):

$$f: E \times Q \to P(Q)$$

Es decir, $f(e,q) = \{q_a, q_b, \ldots . q_l\} \subset Q$. (Obsérvese que puede ser $f(e,q) = \emptyset$).

Un reconocedor finito no determinista se define siguiendo la misma línea que en el caso determinista: se considera un conjunto de estados finales, F, un estado inicial, q_1, y

$$RFND = \langle E, Q, f, q_1, F \rangle$$

El dominio de la función de transición puede extenderse a $E^* \times Q$: Recuérdese que en el caso determinista hacíamos - (Tema "Autómatas", Cap. 2, Ap. 3.2) $f(\Lambda,q) = q$; $f(x_1x_2,q) =$

$= f(x_2, f(x_1, q)$; ahora haremos

$$f(\Lambda, q) = \{q\};\quad f(x_1 x_2, q) = \bigcup_{q_k \text{ en } f(x_1,q)} f(x_2, q_k)$$

También puede extenderse tal dominio a $E^* \times P(Q)$; para ello haremos:

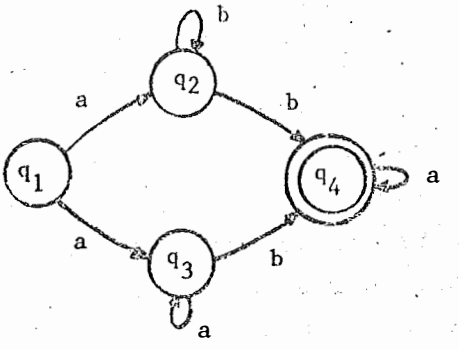
$$f(x, \emptyset) = \emptyset$$

$$f(x, \{ q_a, q_b \ldots . q_l \}) = \bigcup_{j=a}^{l} f(x, q_j)$$

Ejemplo 5.1.2. Sea el RFND definido por

$E = \{a,b\}$; $Q = \{q_1, q_2, q_3, q_4\}$; $F = \{q_4\}$

	a	b
q_1	$\{q_2, q_3\}$	$\emptyset$
q_2	$\emptyset$	$\{q_2, q_4\}$
q_3	$\{q_3\}$	$\{q_4\}$
(q_4)	$\{q_4\}$	$\emptyset$



Estudiemos, por ejemplo, las transiciones producidas por la cadena x = abbab

$$f(a, q_1) = \{q_2, q_3\}$$

$$f(ab, q_1) = f(b, q_2) \cup f(b, q_3) = \{q_2, q_4\}$$

$$f(abb, q_1) = f(b, q_2) \cup f(b, q_4) = \{q_2, q_4\}$$

$$f(abba, q_1) = f(a, q_2) \cup f(a, q_4) = \{q_4\}$$

$$f(abbab, q_1) = f(b, q_4) = \emptyset$$

Si quisiéramos calcular, por ejemplo, $f(a, \{q_1,q_4\})$, - teniendo en cuenta cómo se ha definido la extensión a $E^* \times P(Q)$, tendríamos:

$$f(a,\{q_1,q_4\}) = f(a,q_1) \cup f(a,q_4) = \{q_2,q_3,q_4\}$$

5.2. Lenguaje aceptado por un reconocedor finito no determinista, y equivalencia con algún reconocedor finito determinista.

Definiremos el lenguaje aceptado por un RFND como el - conjunto de cadenas para las cuales la función de transición - conduce a un subconjunto de Q dentro del cual se encuentra al menos un estado final:

$$L(RFND) = \{x \in E^* \mid f(x,q_1) = \{q_a,q_b,\ldots q_l\} \quad y$$

$$\exists q_i (i = a \ldots . l) \in F\}$$

Así, en el ejemplo anterior vemos que la cadena *a* es - rechazada, *ab*, *abb* y *abba* son aceptadas, y *abbab* es rechazada. Es fácil ver que el lenguaje admite una expresión regular: - - ab^*ba^* es la expresión regular del conjunto de cadenas que, pasando por el estado q_2, son aceptadas, mientras que aa^*ba^* es la de las que pasan por q_3; por tanto, la expresión regular de todas será: $ab^*ba^* + aa^*ba^* = a(a^* + b^*)\,ba^*$.

Pasaremos ahora a ver que la clase de los lenguajes - aceptados por reconocedores finitos no deterministas coincide con la de los aceptados por reconocedores finitos determinis-- tas. Para ello será preciso demostrar dos cosas:

- que, dado un RF (determinista), siempre existe un - RFND tal que L(RFND) = L(RF), (y, por tanto,

$$\{L(RF)\} \subset \{L(RFND)\}, \quad y$$

- que, dado un RFND, siempre existe un RF tal que - - L(RF) = L(RFND) (y, por tanto, {L(RFND)C{L(RF)}, que, con lo anterior, demostrará que {L(RFND)} = {L(RF)}).

Lo primero es evidente, ya que un RF es un caso particular de RFND en el que los elementos del rango de f son subconjuntos unitarios de P(Q). Lo segundo no es tan evidente, - por lo que pasaremos a enunciarlo en forma de teorema y demostrarlo.

Teorema 5.2.1. Para todo reconocedor finito no determinista, RFND = $<E,Q,f,q_1,F>$, puede construirse un reconocedor - finito determinista, RF = $<E,Q',f'_1,q'_1,F'>$, tal que L(RF) = - = L(RFND).

Hagamos: Q' = P(Q) (de modo que RF tendrá, en general, $card(Q') = 2^{card(Q)}$ estados). Al estado de Q' que corresponda a $\{q_a,q_b,....q_l\}$ lo denotaremos $[q_a,q_b,....q_l]$.

$$f'(e, [q_a,...q_l]) = [q_m,...q_k] \text{ si y sólo si}$$

$$f(e, \{q_a,....q_l\}) = \{q_m,...q_k\}.$$

(Es decir, se calcula f' (e,q') aplicando f a cada estado q de los que figuran en q' y haciendo la unión de todos los resul-- tantes).

$$q'_1 = [q_1]$$

$$F' = \{q' \in Q' \mid q_f \in q' \text{ y } q_f \in F\}$$

(Es decir, para que q' sea estado final basta que uno o más de los estados de Q que lo componen sea final).

Para demostrar que ambos autómatas aceptan el mismo - lenguaje bastará comprobar que, para todo $x \in E^*$, $f'(x,q_1) \in F'$

si y sólo si $f(x,q_1)$ contiene un estado (o varios) $q_f \in F$, y, teniendo en cuenta la definición de F', esto será evidentemente cierto si demostramos que

$$f'(x,q_1') = [q_a,\ldots q_1] \quad \text{si y sólo si } f(x,q_1) = \{q_a,\ldots.q_\ell\}$$

Tal demostración puede hacerse por inducción sobre la longitud de x: Para $lg(x) = 0$ $(x = \Lambda)$ es inmediato, puesto que $f'(\Lambda,q_1') = q_1' = [q_1]$, y $f(\Lambda,q_1) = \{q_1\}$. Supongamos que es cierto para $lg(x) \leq l$; entonces para $e \in E$,

$$f'(xe, q_1') = f'(e,f'(x,q_1')$$

Pero por la hipótesis de la inducción,

$$f'(x,q_1') = [q_a,\ldots q_1] \text{ si y sólo si } f(x,q_1) = \{q_a,\ldots.q_1\},$$

y, por definición de f',

$$f'(e,[q_a,\ldots,q_1]) = [q_m,\ldots.,q_k] \text{ si y sólo si}$$

$$f(e, \{q_a,\ldots.,q_1\}) = \{q_m,\ldots q_k\}.$$

Por tanto,

$$f'(xe,q_1') = [q_m,\ldots,q_k] \text{ si y sólo si } f(xe,q_1) = \{q_m,\ldots,q_k\},$$

con lo que queda demostrado.

Ejemplo 5.2.2.

Tomemos el RFND del ejemplo 5.1.2. Siguiendo la construcción del Teorema 5.2.1, el RF tendrá, en principio, $2^4 = 16$ estados:

$$Q' = \{\emptyset, [q_1], [q_2], [q_3], [q_4], [q_1,q_2] \ldots\ldots [q_1,q_2,q_3,q_4]\}$$

$q'_1 = [q_1]$

$F' = \{[q_4], [q_1,q_4], [q_2,q_4], [q_3,q_4], [q_1,q_2,q_4], \ldots\ldots$

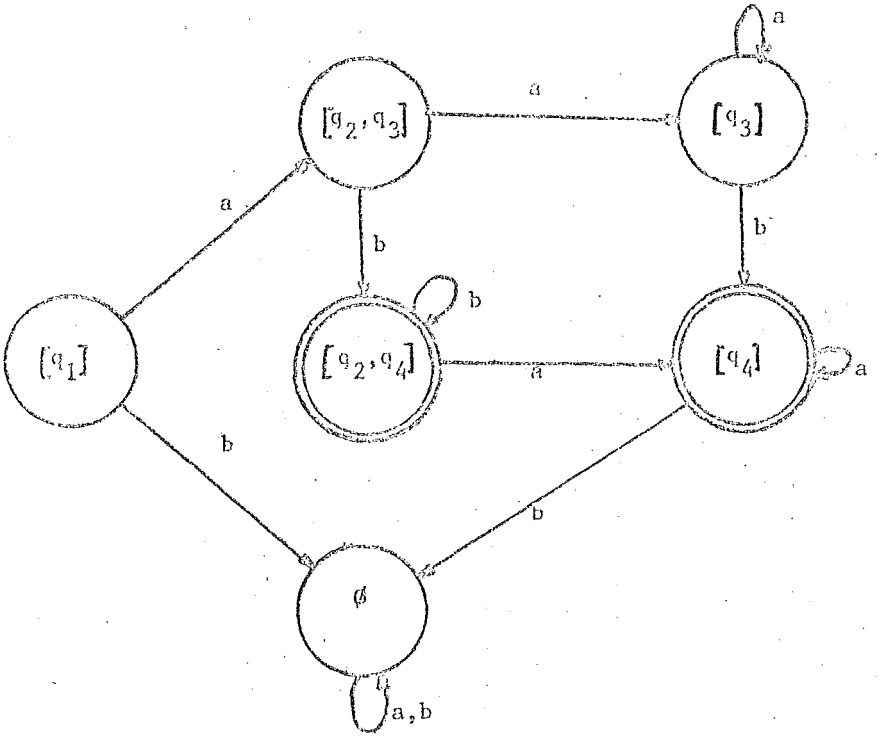
$\ldots\ldots [q_1,q_2,q_3,q_4]\}$

y f' vendrá dada por la siguiente tabla:

	a	b	
$\emptyset$	$\emptyset$	$\emptyset$	
$[q_1]$	$[q_2,q_3]$	$\emptyset$	
$[q_2]$	$\emptyset$	$[q_2,q_4]$	←
$[q_3]$	$[q_3]$	$[q_4]$	
$[q_4]$	$[q_4]$	$\emptyset$	
$[q_1,q_2]$	$[q_2,q_3]$	$[q_2,q_4]$	←
$[q_1,q_3]$	$[q_2,q_3]$	$[q_4]$	←
$[q_1,q_4]$	$[q_2,q_3,q_4]$	$\emptyset$	←
$[q_2,q_3]$	$[q_3]$	$[q_2,q_4]$	
$[q_2,q_4]$	$[q_4]$	$[q_2,q_4]$	
$[q_3,q_4]$	$[q_3,q_4]$	$[q_4]$	←
$[q_1,q_2,q_3]$	$[q_2,q_3]$	$[q_2,q_4]$	←
$[q_1,q_2,q_4]$	$[q_2,q_3,q_4]$	$[q_2,q_4]$	←
$[q_1,q_3,q_4]$	$[q_2,q_3,q_4]$	$[q_4]$	←
$[q_2,q_3,q_4]$	$[q_3,q_4]$	$[q_2,q_4]$	←
$[q_1,q_2,q_3,q_4]$	$[q_2,q_3,q_4]$	$[q_2,q_4]$	←

Ahora bien, en un RF los estados que no son accesibles desde el inicial pueden eliminarse. Así, eliminamos $[q_2]$ porque

no lo vemos aparecer dentro de la tabla, y lo mismo $[q_1,q_2]$, - etc. (los señalados con una flecha). Obsérvese que $[q_2,q_3,q_4]$ sólo es accesible desde otros que previamente han sido elimina ods, por lo que también puede eliminarse, y lo mismo ocurre - con $[q_3,q_4]$. Naturalmente, el único del que no puede prescin-- dirse en ningún caso es de $|q_1|$. Ø, en este caso, no puede eli minarse, ya que es accesible desde $[q_1]$ (y desde $[q_4]$). Final- mente, de los 16 estados nos hemos quedado con 6, y el resulta do puede representarse en forma de diagrama de Moore:



Comprobemos que el lenguaje aceptado es el mismo del - RFND de partida (expresión regular $a(a^* + b^*)ba^*$). Al estado - final $[q_2,q_4]$ sólo podemos ir pasando por $[q_2,q_3]$; tenemos así que las cadenas aceptadas en $[q_2,q_4]$ tienen por expresión regu lar: abb^*; a $[q_4]$ puede llegarse por $[q_2,q_4]$ (abb^*aa^*) o por - $[q_3]$ (aaa^*ba^*). Tenemos, pues, en total:

$$abb^* + abb^*aa^* + aaa^*ba^* = abb^*(\Lambda+aa^*) + aaa^*ba^* =$$

$$= abb^*a^* + aaa^*ba^* = ab^*ba^* + aaa^*ba^* =$$

$$= a(b^* + aa^*)ba^* = a(b^* + a^*)\, ba^*$$

(Esta última igualdad se ha obtenido teniendo en cuenta que $b^* + aa^* = b^* + \Lambda + aa^* = b^* + a^*$). Naturalmente, llegaríamos al mismo resultado aplicando el algoritmo general de análisis desarrollado en el Tema "Autómatas", Cap. 4, Ap. 4.

5.3. Teorema AF1.

Para toda gramática regular, G3, existe un reconocedor finito, RF, tal que L(RF) = L(G3).

Para demostrar este teorema construiremos un RFND que reconoce exactamente el lenguaje aceptado por una G3 dada, al que le corresponderá un RF de acuerdo con el teorema 5.2.1.

Sea G3 = $<E_A, E_T, P, S>$ una gramática regular. Definimos RFND = $<E, Q, f, q_1, F>$ del siguiente modo:

$E = E_T = \{a_1, a_2, \ldots . a_n\}$

$Q = E_A \cup \{X\} = \{A_1, A_2, \ldots, A_n, X\}$

$q_1 = S$

$F = \{X\}$

Si $(A_i \rightarrow a_j) \in P$, $X \in f(a_j, A_i)$

Si $(A_i \rightarrow a_j A_k) \in P$, $A_k \in f(a_j, A_i)$

$f(a_i, X) = \emptyset \quad \forall a_i \in E_T$

Veamos ahora que L(RFND) = L(G3)

a) Sea $x = a_1a_2....a_k \in L(G3)$; entonces, deberán existir $A_1,A_2....,A_{k-1} \in E_A$ tales que

$$S \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow \ldots \Rightarrow a_1a_2....a_{k-1}A_{k-1} \Rightarrow a_1a_2....a_{k-1}a_k$$

y para ello P debe contener las reglas

$$\begin{aligned} S &\rightarrow a_1A_1 \\ A_1 &\rightarrow a_2A_2 \\ &\vdots \quad \vdots \\ A_{k-1} &\rightarrow a_k \end{aligned}$$

Por la definición de f, $A_1 \in f(a_1,q_1)$; $A_2 \in f(a_2,A_1)$; ....$X \in f(a_k,A_{k-1})$; por consiguiente, $X \in f(a_1a_2....a_k,q_1)$, y, como X es el estado final, $a_1a_2....a_k \in L(RFND)$. Esto nos dice que $L(G3) \subset L(RFND)$.

b) A la inversa, si $x = a_1a_2.....a_k \in L(RFND)$, deberá existir una secuencia de estados $A_1,A_2,....A_{k-1},X$, tal que

$$\begin{aligned} A_1 &\in f(a_1,q_1) \\ A_2 &\in f(a_2,A_1) \\ &\vdots \quad \vdots \\ A_{k-1} &\in f(a_{k-1},A_{k-2}) \\ X &\in f(a_k,A_{k-1}) \end{aligned}$$

y, conforme a la construcción del RFND, G3 deberá tener las reglas: $S \rightarrow a_1A_1$; $A_1 \rightarrow a_2A_2$;...$A_{k-1} \rightarrow a_k$, con las cuales se podrá hacer la derivación $S \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow \ldots \Rightarrow a_1a_2....a_k$, es decir, $a_1a_2....a_k \in L(G3)$. Es decir, $L(RFND) \subset L(G3)$.

En resumen, $L(RFND) = L(G3)$.

Ejemplo 5.3.1. Tomemos la gramática regular del Capítulo 2, Ap. 4. Ej. 5:

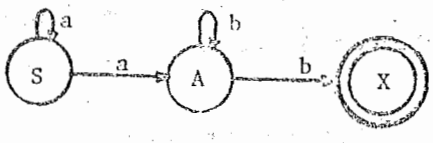
$$E_A = \{A,S\}; \quad E_T = \{a,b\};$$

$$P = \left\{ \begin{array}{l} S \to aS \\ S \to aA \\ A \to bA \\ A \to b \end{array} \right\}$$

El correspondiente RFND será:

$$E = E_T = \{a,b\}; \; Q = \{A,S,X\}; \; F = \{X\}; \; q_1 = S$$

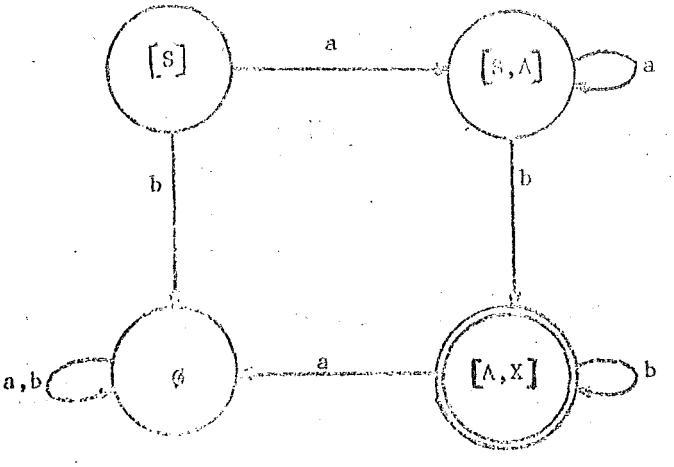
	a	b
S	{S,A}	Ø
A	Ø	{A,X}
X	Ø	Ø



Podemos construir un RF que acepte el mismo lenguaje, de acuerdo con el teorema 5.2.1.:

	a	b	
Ø	Ø	Ø	
[S]	[S,A]	Ø	
[A]	Ø	[A,X]	←
[X]	Ø	Ø	←
[S,A]	[S,A]	[A,X]	
[S,X]	[S,A]	Ø	←
[A,X]	Ø	[A,X]	
[S,A,X]	[S,A]	[A,X]	←

Eliminamos los estados [A], [X], [S,X], [S,A,X], inaccesibles desde [S], y resulta el siguiente diagrama de Moore:



Es fácil comprobar que el lenguaje viene dado por la expresión regular aa^*bb^*.

5.4. Teorema AF2.

Para todo reconocedor finito, RF, existe una gramática regular, G3, tal que L(G3) = L(RF).

Dado RF = $<E, Q, f, q_1, F>$, construiremos G3 = $<E_A\ E_T, P, S>$ así:

$$E_A = Q;\ E_T = E;\ S = q_1$$

Si $f(a,q_i) = q_j$, $q_i \rightarrow aq_j$

Si $f(a,q_i) = q_j$ y $q_j \in F$, $q_i \rightarrow a$ (además de $q_i \rightarrow aq_j$)

Si $f(a,q_i) = q_i\ \forall a$ y $q_i \notin F$, puede eliminarse q_i y todas las reglas en las que figure.

La demostración de que L(G3) = L(RF) se hace viendo -

que $(q_1 \overset{*}{\underset{G_3}{\Rightarrow}} x)$ si y sólo si $(f(x,q_1) \in F)$, $\forall x \in E^*$, de manera similar a la del teorema anterior.

Ejemplo 5.4.1. Consideremos el RF al que llegábamos en el ejemplo 5.3.1 y retrocedamos ahora en busca de la gramática. Haciendo, para simplificar la escritura, $[S] = S$, $[S,A] = A$, $[A,X] = B$, tendremos:

$$E_A = \{S,A,B\}; \quad E_T = \{a,b\}$$

(Obsérvese que no hemos incluido $\emptyset$ en E_A porque $f(a,\emptyset) = f(b,\emptyset) = \emptyset$). Siguiendo las normas para la construcción de las reglas obtenemos:

$$S \to aA$$
$$A \to aA$$
$$A \to bB$$
$$A \to b$$
$$B \to bB$$
$$B \to b$$

Esta gramática tiene un símbolo auxiliar (B) y dos reglas más que aquélla de la que habíamos partido, aunque, naturalmente, ambas deben ser equivalentes. La diferencia se ha originado en el paso a través del RF determinista. El lector puede extender la construcción del teorema AF2 al caso más general de RFND, y aplicarla al ejemplo 5.3.1 para ver que entonces sí se llega a la misma gramática original.

5.5. Conclusión.

Según el teorema AF1, $\{L(G3)\} \subset \{L(RF)\}$, y según el AF2, $\{L(RF)\} \subset \{L(G3)\}$. Además, del Tema "Autómatas" (Cap. 4, Ap. 4.1.1; teorema de análisis) sabemos que $\{L(RF)\} = \{L_{regulares}\}$. En resumen:

$$\{L_{regulares}\} = \{L(RF)\} = \{L(G3)\}$$

5.6. Observaciones sobre la cadena vacía.

En el teorema AF1 hemos supuesto implícitamente que - $\Lambda \notin L(G3)$. Si $\in L(G3)$, entonces, según se vio en el Capítulo anterior, existirá la regla $S \to \Lambda$ (y S no aparecerá a la derecha de ninguna otra regla). En este caso, la única diferencia es que habrá que considerar S también como estado final: F = = {S,X}.

En cuanto al caso inverso (teorema AF2), si en el RF - de partida $q_1 \in F$, entonces $\Lambda \in L(RF)$, y $L(G3) = L(RF) - \{\Lambda\}$. Según vimos, puede construirse G'3 tal que

$$L(G'3) = L(G3) \cup \{\Lambda\} = L(RF)$$

6. JERARQUÍA DE AUTÓMATAS.

Paralelamente a la jerarquía de lenguajes, aparece una jerarquía de autómatas, desde la MT hasta el AF. Para comparar los puede utilizarse como criterio la capacidad de memoria de cada uno.

La MT dispone, sobre la cinta, de una capacidad ilimitada de casillas para memorizar una cantidad ilimitada de símbolos.

La cinta de un ALL es también ilimitada, pero para una determinada cadena de entrada, x, de longitud l = lg(x), sólo puede utilizar l casillas. Además, habrá que añadir a esta capacidad variable de memoria (dependiente, en cada caso, de cada entrada), una capacidad fija, n, que corresponde a la capacidad de la unidad central para memorizar estados. En resumen, la memoria de un ALL es una función de la longitud l de la cadena de entrada: l + n. La forma lineal de esta función justifica el nombre del autómata.

El AP sólo puede utilizar la cinta de entrada para - - leer, pero dispone de una cinta de trabajo o pila potencialmente ilimitada que le sirve de memoria. Aquí también puede estimarse, para una longitud l de la cadena de entrada, un límite superior de la memoria a utilizar. En efecto, si la cadena más larga que puede escribirse en la pila como consecuencia de la lectura de un símbolo de entrada tiene longitud k, la capacidad total de memoria será la suma de la parte variable (pila): kl y de la parte fija, n; en total, kl + n. Esta es también - una función lineal de l, por lo que del AP puede también decirse que es un autómata limitado linealmente, pero con dos restricciones importantes: la cinta de entrada sólo se desplaza - en un sentido, y la memoria tiene estructura de pila (si se - quiere recuperar un símbolo que no está en la cima, hay que borrar toda la información comprendida entre la cima y el símbolo, cosa que no ocurre con la MT ni el ALL).

La capacidad de memoria de un AF es fija e independiente de la cadena de entrada.

Finalmente, es preciso que mencionemos un punto importante que, al omitir la demostración de la mayoría de los teoremas, hemos pasado por alto: que los autómatas de que venimos hablando son, en general, no deterministas, es decir, que la - función de transición de la unidad de control tiene la forma - que veíamos en el AF no determinista. Esto solamente serviría para demostrar los teoremas, y no tendría mayor importancia, - si en todos los tipos de reconocedores ocurriese lo que en el RF: que para todo RFND puede encontrarse un RF que acepte el - mismo lenguaje, pero no es así:

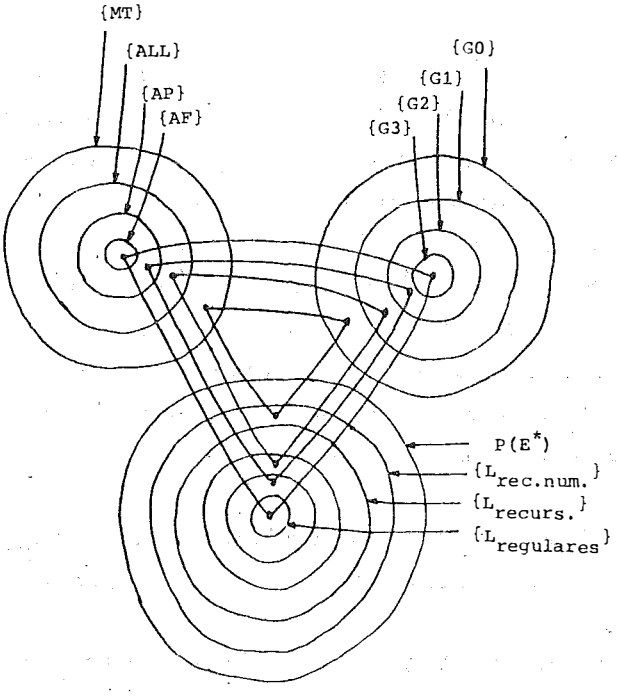
a) En la MT, se demuestra que $\{L(R_{MT})\} = \{L(R_{MT}ND)\}$, - es decir, ocurre lo mismo que en el RF.

b) En el ALL *no se sabe* si $\{L(R_{ALL})\} = \{L(R_{ALL}ND)\}$ o

$\{L(R_{ALL})\} \subset \{L(R_{ALL}ND)\}$. Por tanto, cuando se habla de equivalencia de lenguajes sensibles al contexto y lenguajes aceptados por R_{ALL} debe entenderse que éstos son *no deterministas*. Por supuesto, para todo $R_{ALL}ND$ puede diseñarse un R_{MT} que acepte el mismo lenguaje; lo único que ocurre es que la longitud de cinta necesaria en el R_{MT} (determinista) puede ser una función exponencial de la longitud de la cadena de entrada, en lugar de lineal, como en el $R_{ALL}ND$.

c) En el AP, *se sabe* que $\{L(R_{AP})\} \subset \{L(R_{APND})\}$, y, es más, se conoce el tipo de gramáticas correspondientes a los R_{AP} deterministas, que son una particularización del tipo general de gramáticas de contexto libre, llamadas *gramáticas deterministas*, y que, entre otras propiedades, no son ambiguas.

7. RESUMEN.



8. Notas histórica y bibliográfica.

En los Temas "Autómatas" y "Algoritmos" se han dado referencias históricas sobre los AF y las MT, respectivamente. El concepto de AF no determinista se debe a RABIN y SCOTT (1959), aunque las relaciones entre AF y gramáticas regulares ya habían sido establecidas por CHOMSKY y MILLER (1958). La equivalencia entre lenguajes aceptados por MT y lenguajes de tipo 0 fue demostrada por CHOMSKY (1959).

El nombre y el concepto de ALL fueron introducidos por MYHILL (1960). La generalización al caso no determinista y la equivalencia con las gramáticas sensibles al contexto se deben a KURODA (1964).

OETTINGER (1961) fue el primero en definir formalmente el AP, y CHOMSKY (1962) y EVEY (1963), independientemente, demostraron su relación con los lenguajes de contexto libre.

Como bibliografía de consulta, son recomendables los mismos libros citados en los anteriores Capítulos, y, especialmente, el de HOPCROFT y ULLMAN (1969).

9. Ejercicios.

1. Hallar una gramática regular que genere el lenguaje correspondiente al RF del ejemplo 5.2.2. Partir del RFND del ejemplo 5.1.2 y obtener otra gramática, equivalente pero más sencilla.

2. Hallar gramáticas regulares correspondientes a los cuatro ejemplos que se utilizaron en el Capítulo 4 del Tema "Autómatas".

3. Dada la expresión regular

$$\alpha = 00^* + 10^*10^*,$$

obtener un reconocedor finito del correspondiente lenguaje, y una gramática regular que lo genere.

4. Dada la gramática regular definida por:

$$E_A = \{S,A,B,C\}; \quad E_r = \{0,1\};$$

$$P = \left\{ \begin{array}{l} S \to 1A|1B \\ A \to 0A|0C|1C|1 \\ B \to 1A|1C|1 \\ C \to 0 \end{array} \right\}$$

dar una expresión regular del lenguaje generado por ella y diseñar un circuito secuencial con biestables JK que sirva como reconocedor de tal lenguaje.

5. Repetir el ejercicio anterior para la gramática:

$$E_A = \{S,A\}; \quad E_T = \{a,b,c,d\};$$

$$P = \left\{ \begin{array}{l} S \to aS|aA|bA|cA|dA|a|b|c|d \\ A \to aA|bA|a|b \end{array} \right\}$$

6. (HOPCROFT y ULLMAN, 1969). Utilizar el concepto de RFND para demostrar que, si L es un lenguaje regular,

$$L^R = \{x \mid x^R \in L\}$$

es también regular

(x^R significa "reflejada" de x, es decir, si

$x = a_1a_2\ldots a_{n-1}a_n$, $x^R = a_na_{n-1}\ldots a_2a_1$).

7. (HOPCROFT y ULLMAN, 1969). Apoyándose en el ejercicio an

terior, demostrar que los lenguajes generados por gramáticas cuyas reglas son de la forma $A \rightarrow Ba$, ó $A \rightarrow a$ ($a \in E_T$; $A,B, \in E_A$) son lenguajes regulares (y viceversa: todo lenguaje regular puede generarse por una gramática de ese tipo).

CAPITULO 5.

NOCIONES SOBRE TRADUCTORES DE LENGUAJES

1. OBJETIVOS Y CONTENIDO.

Del Tema "Estructura y funcionamiento de los ordenadores" el alumno conoce las funciones de, y diferencias entre los distintos traductores de lenguajes de programación (ensambladores, compiladores e intérpretes) y tiene noticia de la existencia y cometidos de los sistemas operativos.

En este Capítulo nos proponemos dar unas ideas básicas sobre las técnicas que se utilizan en la construcción de traductores. La exposición ha de ser necesariamente resumida y de tipo general, sin poder entrar en detalles, y aun los ejemplos concretos no serán más que modelos simplificados de los lenguajes de programación reales.

Veremos en primer lugar los ensambladores. El procesamiento en este caso es relativamente sencillo, y, aunque no lo expondremos detalladamente, el alumno que tenga una mínima experiencia en programación no encontraría especiales dificultades en escribir un ensamblador para cualquier máquina. Los compiladores e intérpretes ya son más complicados; nos contentaremos con que se intuyan las posibilidades de aplicación de los conceptos estudiados en Capítulos anteriores a la programación de las diferentes fases.

2. ENSAMBLADORES.

2.1. El proceso de ensamblaje. Ensambladores de dos y de un paso.

La mayoría de los ensambladores son de dos pasos, es - decir, leen dos veces el programa fuente. La necesidad de ha-- cer así viene impuesta por el hecho de que es imposible determinar una dirección absoluta si la etiqueta correspondiente - aparece más adelante. Por ejemplo, consideremos el siguiente - segmento de un programa en ENSAM:

```
          SAI     PRINC

UNØ       CEN     1
N         REM     1
A         REM     100
PRINC     LNE     N
            .
            .
            .
```

Al pretender traducir la instrucción "SAI PRINC", no podemos saber cuál es la dirección absoluta, que dependerá del número de instrucciones y pseudoinstrucciones que hay entre - "SAI PRINC" y la referenciada por la etiqueta "PRINC".Por ello, lo que se hace es, en una primera lectura total del programa - fuente (primer paso), formar en la memoria una tabla de etiquetas; en el ejemplo anterior, esta tabla quedaría así:

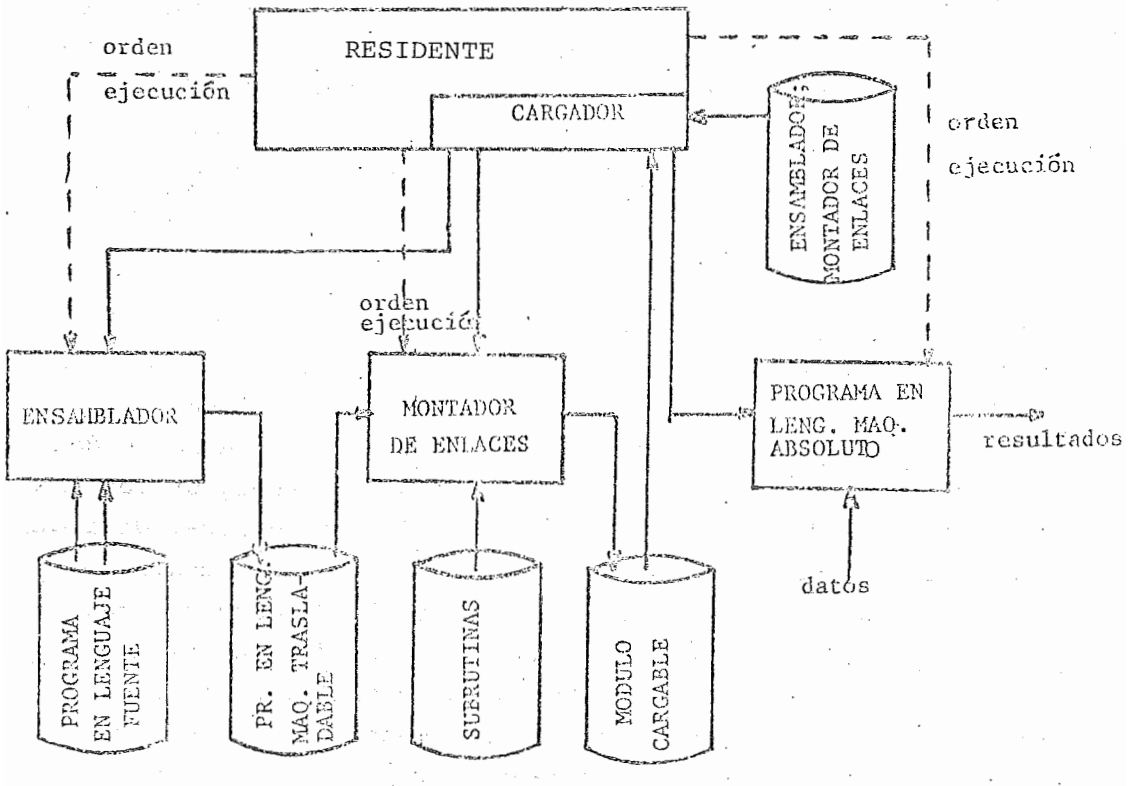
ETIQUETA	DIRECCION
UNØ	1
N	2
A	3
PRINC	103
⋮	⋮

Cuando se ha leído todo el programa, se empieza de nuevo (segundo paso). Al leer entonces "SAI PRINC", se consultará la tabla de etiquetas, donde se verá que a "PRINC" le corresponde la dirección 103, con lo que ya se podrá traducir la instrucción a binario.

Sin embargo, en algunas instalaciones de ordenador es preciso idear algún procedimiento con el que sólo se lea una vez el programa fuente (ensamblador de un paso); la necesidad se comprenderá examinando los procesos que intervienen en el ordenador desde la traducción hasta la ejecución de un programa.

Por limitaciones de capacidad, no siempre es posible que coexistan en la memoria central el programa fuente, el ensamblador y el programa objeto. Por esta razón, el ensamblador siempre está construido de tal modo que durante el ensambaje *sólo él* reside en la memoria central (junto con una parte del sistema operativo, llamada *residente*). El ensamblador leé las instrucciones del programa fuente, una detrás de otra, de algún periférico, y deposita el programa objeto en algún medio de almacenamiento auxiliar (disco, cinta, etc.). Si este programa objeto es ya ejecutable,el *cargador* ("loader"), que es una parte del residente, lo lleva a memoria central (pudiendo ya desaparecer de la misma el ensamblador). Generalmente se hace preciso un proceso intermedio, debido a que el programa objeto necesita otros programas para poder ejecutarse (subrutinas ensambladas independientemente, o rutinas del sistema operativo, como las de entrada/salida);este proceso es llevado a cabo por un programa del sistema operativo llamado *montador de enlaces* ("link editor"). La figura de la página siguiente ilustra esta secuencia de procesos, suponiendo que el almacenamiento auxiliar es sobre disco. En cada momento, en la memoria central sólo se encuentra, aparte del residente, un programa: el ensamblador, el montador de enlaces o el programa objeto. No debe -

interpretarse erróneamente el que en la figura aparezca cinco veces el símbolo de disco: los distintos programas pueden es-- tar en varios discos o en uno solo, en diferentes zonas o fi-- cheros.



Si el programa fuente está escrito en fichas perforadas, el sistema operativo se encargará de leerlo y almacenarlo en un fichero del disco, de donde lo tomará el ensamblador, leyéndolo dos veces, una por cada paso (por eso en la figura aparecen dos flechas de entrada del programa fuente al ensamblador); estas lecturas del disco se hacen sin intervención - del operador. También puede ser el propio primer paso del ensamblador el que lee las fichas y las introduce en disco.

Este es el procedimiento normal de trabajo en una instalación que dispone de disco(s). Con cintas, tambores, etc.,

sería similar. Pero supongamos que la instalación no tiene ningún dispositivo de almacenamiento auxiliar. En tal caso, para que el ensamblador pudiera leer dos veces el programa fuente (suponiendo que éste no se cargue en memoria central) sería preciso que el operador introdujera manualmente dos veces las fichas (o la cinta perforada, o cualquier otro medio en el que esté escrito el programa fuente). Aquí, por tanto, interesa disponer de un ensamblador de un solo paso, aunque ocupe algo más de memoria, ya que debe ir formando una tabla adicional con los nombres no referenciados en la tabla de etiquetas y la dirección de memoria de la correspondiente instrucción; conforme se va llenando la tabla de etiquetas, se retrocede para traducir a binario las instrucciones pendientes.

2.2. El ensamblador de dos pasos.

La tarea del primer paso consiste, esencialmente, en formar la tabla de etiquetas, llevando, al mismo tiempo, algunas tareas de control de la sintaxis, como detección de etiquetas incorrectas o duplicadas. Para formar la tabla, hay una variable llamada *contador de ensamblaje*, que se inicializa en cero y se incrementa al leer cada instrucción o pseudoinstrucción en un número igual al de posiciones de memoria que ocupa tal instrucción o pseudoinstrucción. Así, al leer una línea, primero se detecta si hay algo en el campo de etiqueta; si hay una etiqueta correcta, se introduce en la tabla junto con el valor actual del contador de ensamblaje. Se analiza luego el código de operación para ver en cuánto hay que incrementar el contador, y así sucesivamente.

En el segundo paso se van traduciendo, una a una, en binario, las instrucciones y pseudoinstrucciones, consultando, cuando es preciso, la tabla de etiquetas.

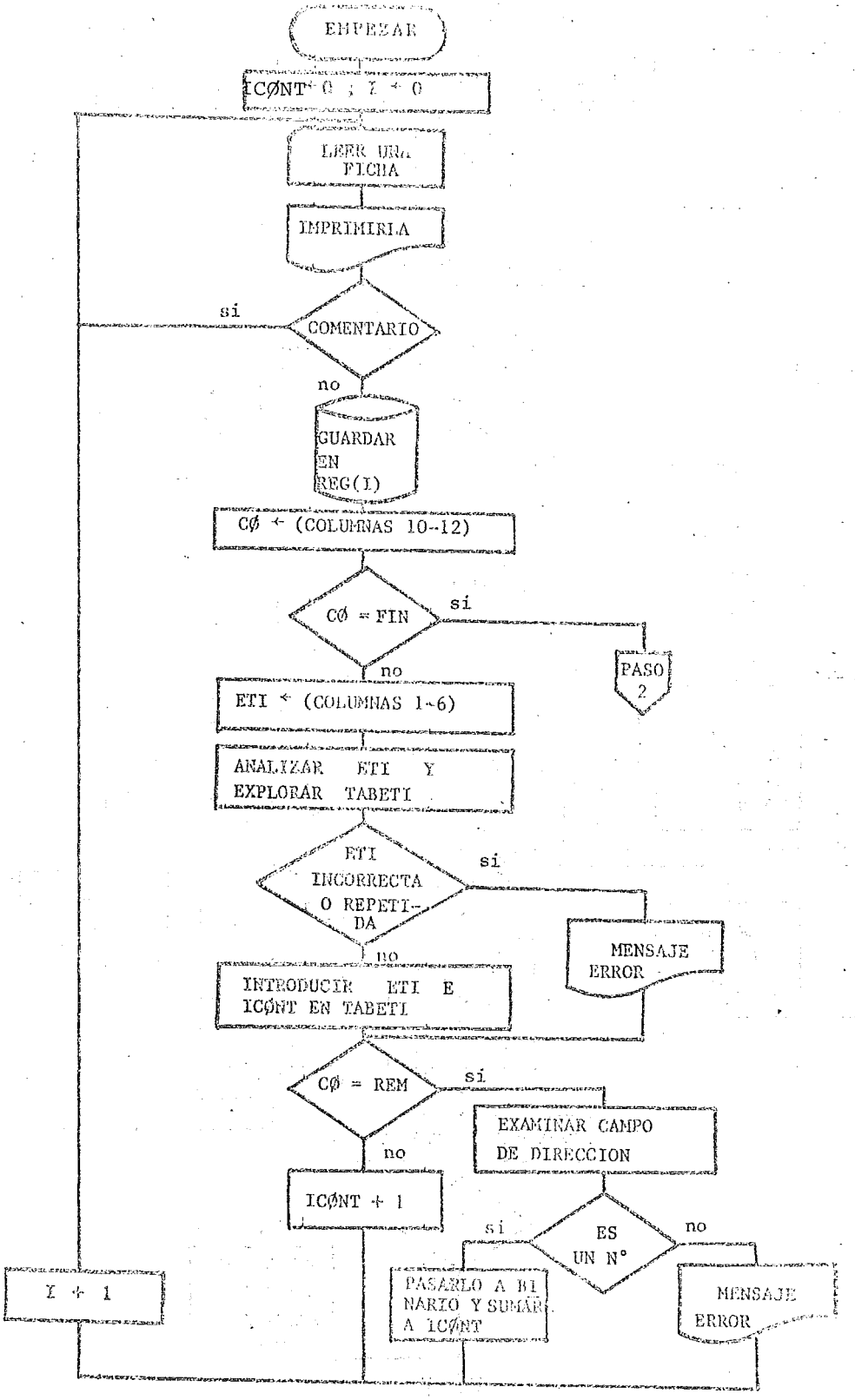
La salida del ensamblador es, fundamentalmente, el pro

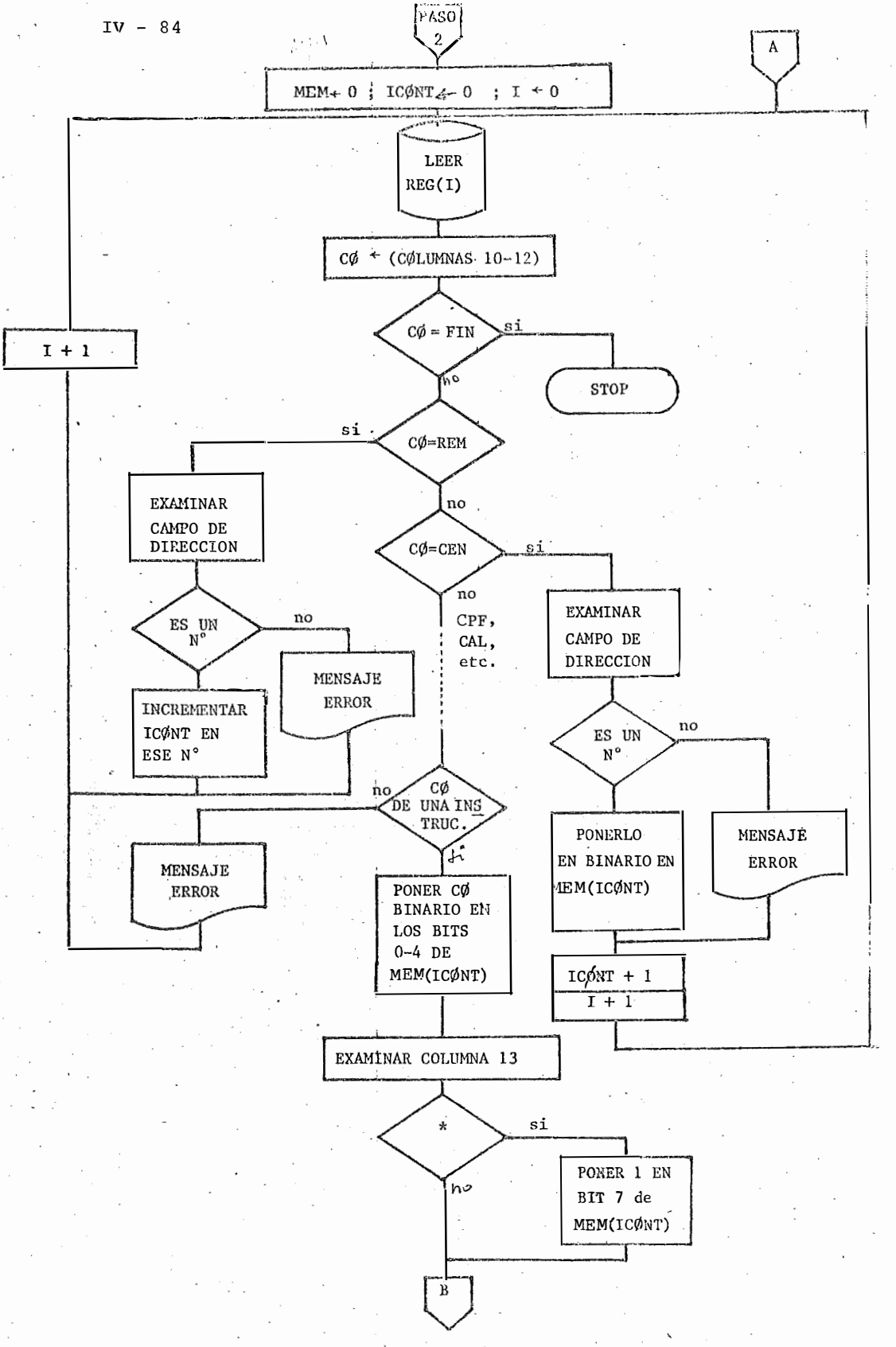
grama binario, pero, además, puede haber referencias a etiquetas externas (como nombres de subprogramas ensamblados independientemente); con tales referencias y la dirección donde aparecen se forma otra tabla que sirve de entrada al montador de enlaces.

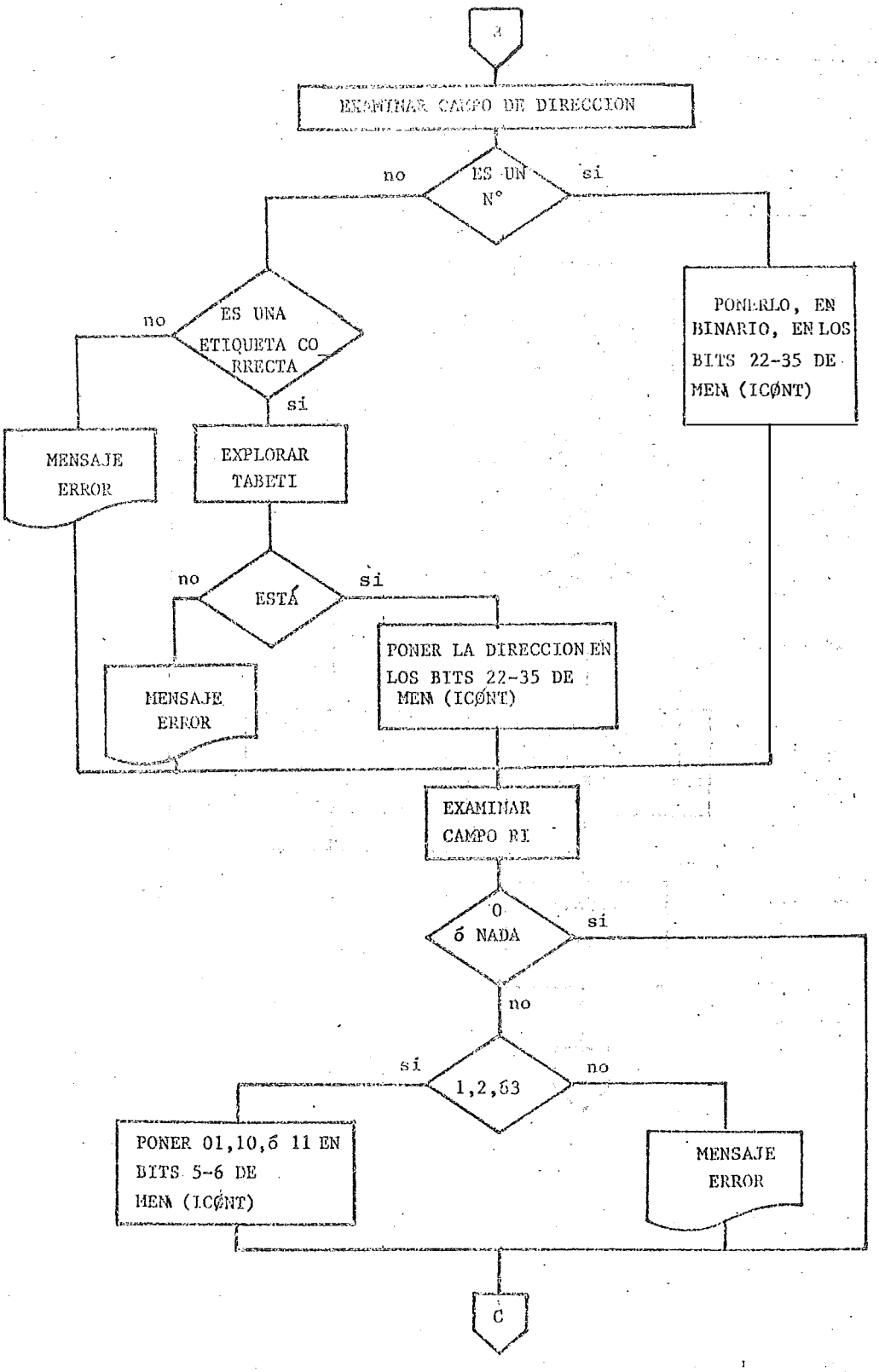
2.3. Ejemplo: Ensamblador de ENSAM.

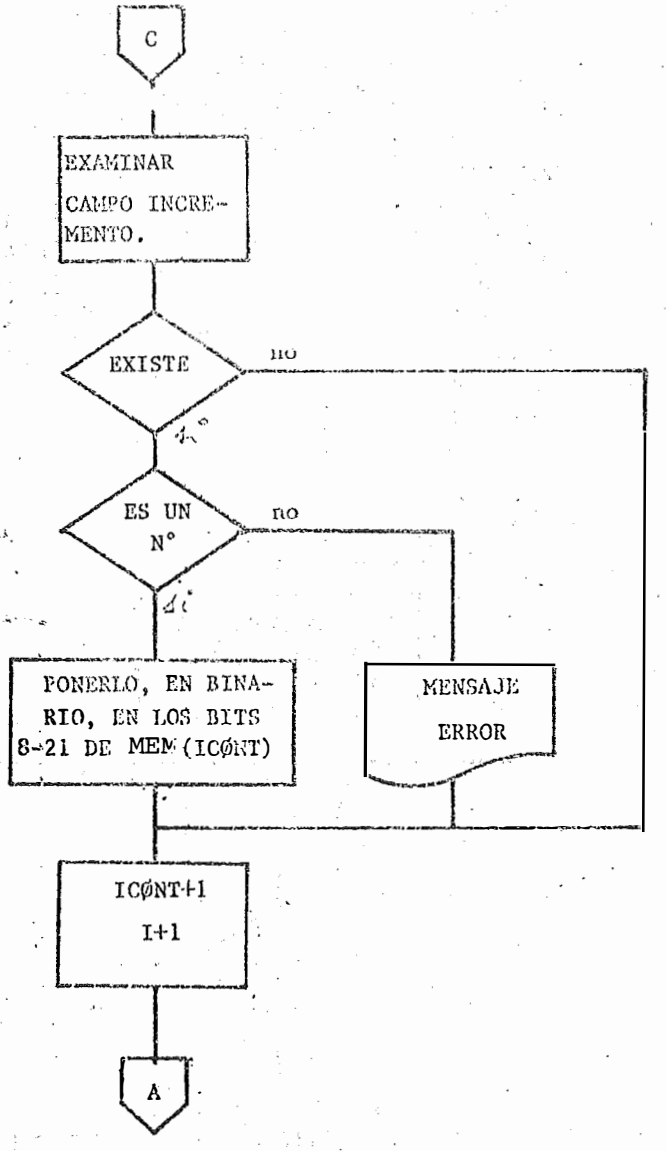
En las páginas siguientes damos unos organigramas generales, no detallados ni completos, de los dos pasos de un ensamblador de ENSAM. ICØNT es el contador de ensamblaje; REG(I) es un vector cuyos elementos deben poder almacenar 80 caracteres; TABETI es la tabla de etiquetas en memoria central. Aunque hemos incluido la detección de casi todos los errores en el paso 2, muchos de ellos pueden detectarse ya en el paso 1 (y es preferible hacerlo así a fin de que el mensaje de error salga listado inmediatamente después del listado de la instrucción y no al final del programa). Se supone que no son posibles las referencias a etiquetas externas, y, por ello, sale un mensaje de error si, en el paso 2, una etiqueta no se encuentra en TABETI.

Aunque, como decíamos más arriba, la salida binaria se hace sobre un soporte (disco, cinta, etc.) para luego cargarse en memoria central, en el caso de ENSAM se deposita directamente sobre memoria para ejecutarse inmediatamente. Realmente, el programa completo de ENSAM tiene, además, un subprograma, al que se llama después de finalizado el paso 2, que simula en el UNIVAC 1108 lo que sería la ejecución en el EIT-2.


EMPEZAR
CØNT ← 0 ; I ← 0
LEER UNA FICHA
IMPRIMIRLA
COMENTARIO
si
no
GUARDAR EN REG(I)
CØ ← (COLUMNAS 10-12)
CØ = FIN
si
PASO 2
no
ETI ← (COLUMNAS 1-6)
ANALIZAR ETI Y EXPLORAR TABETI
ETI INCORRECTA O REPETIDA
si
MENSAJE ERROR
no
INTRODUCIR ETI E ICØNT EN TABETI
CØ = REM
si
EXAMINAR CAMPO DE DIRECCION
no
ICØNT ← 1
ES UN N°
si
no
PASARLO A BINARIO Y SUMAR A ICØNT
MENSAJE ERROR
I ← 1


PASO 2
A
MEM← 0 ; ICØNT← 0 ; I ← 0
LEER REG(I)
CØ ← (CØLUMNAS 10-12)
CØ = FIN
si
STOP
no
CØ=REM
si
EXAMINAR CAMPO DE DIRECCION
ES UN N°
no
MENSAJE ERROR
INCREMENTAR ICØNT EN ESE N°
I + 1
no
CØ=CEN
si
CPF, CAL, etc.
EXAMINAR CAMPO DE DIRECCION
ES UN N°
no
MENSAJE ERROR
PONERLO EN BINARIO EN MEM(ICØNT)
ICØNT + 1
I + 1
no
CØ DE UNA INSTRUC.
si
MENSAJE ERROR
PONER CØ BINARIO EN LOS BITS 0-4 DE MEM(ICØNT)
EXAMINAR COLUMNA 13
*
si
PONER 1 EN BIT 7 de MEM(ICØNT)
no
B

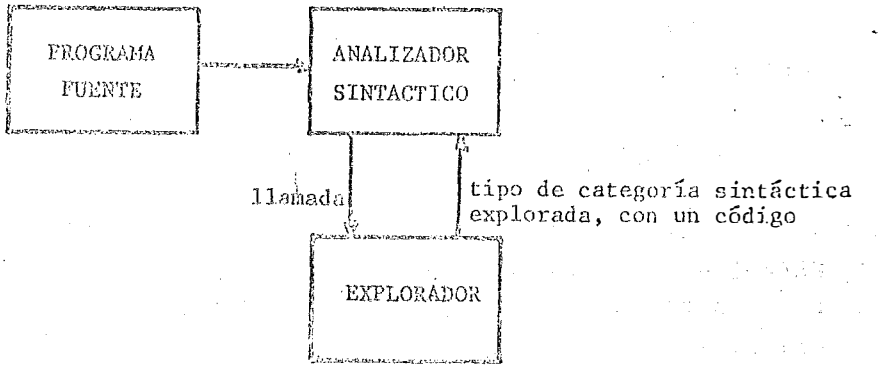

3
EXAMINAR CAMPO DE DIRECCION
ES UN N°
no
si
ES UNA ETIQUETA CORRECTA
no
si
PONERLO, EN BINARIO, EN LOS BITS 22-35 DE MEM (ICØNT)
MENSAJE ERROR
EXPLORAR TABETI
ESTÁ
no
si
MENSAJE ERROR
PONER LA DIRECCION EN LOS BITS 22-35 DE MEM (ICØNT)
EXAMINAR CAMPO RI
0 ó NADA
si
no
1,2,ó3
si
no
PONER 01,10,ó 11 EN BITS 5-6 DE MEM (ICØNT)
MENSAJE ERROR
C


C
EXAMINAR CAMPO INCREMENTO.
EXISTE
no
SI
ES UN N°
no
SI
PONERLO, EN BINARIO, EN LOS BITS 8-21 DE MEM (ICØNT)
MENSAJE ERROR
ICØNT+1
I+1
A

3. COMPILADORES.

3.1. Fases y pasos de un compilador.

En el Tema "Estructura y funcionamiento" se han descrito las distintas fases del proceso de compilación (exploración, análisis sintáctico, análisis semántico, optimización y generación del código). Son estas fases lógicas, que no necesariamente deben ejecutarse en ese orden; de hecho, pueden y suelen simultanearse. Por ejemplo, el explorador es generalmente un subprograma llamado por el analizador sintáctico:



El número de pasos del compilador (lecturas sucesivas del programa fuente o de alguna versión intermedia) es muy variable, y depende de muchos factores. Si la capacidad de memoria central disponible es pequeña interesa, normalmente, realizar el proceso en muchos pasos, con resultados intermedios en memoria auxiliar; lo mismo ocurre si trabajan muchas personas en el diseño: la multiplicación de pasos facilita el desglose de tareas. En el caso extremo que mencionábamos al hablar de los ensambladores, es decir, si no se dispone de ningún medio de almacenamiento auxiliar, es necesario que todo el proceso tenga lugar en un solo paso.

La salida del compilador puede ser en lenguaje de máquina, y entonces los procesos que tienen lugar hasta la ejecución son idénticos a los descritos en el caso del ensamblador, y la misma figura del Apartado 2.1. sigue siendo válida, sustituyendo "ensamblador" por "compilador". Pero hay compiladores cuya salida es el programa en lenguaje ensamblador, por lo que es preciso ejecutar el ensamblador inmediatamente después de la compilación. Finalmente, otros compiladores dan como resultado una secuencia de instrucciones en un lenguaje intermedio, que se ejecutan mediante subprogramas asociados a cada tipo de instrucción, es decir, se ejecutan en *modo interpretativo*.

Pasemos a exponer algunas de las técnicas utilizadas para la programación de las fases.

3.2. Análisis lexicográfico.

El análisis lexicográfico es un análisis al nivel de caracteres, realizado por el *explorador* ("scanner"). Su misión es reconocer las unidades sintácticas (identificadores, delimitadores, constantes, etc.), devolviendo al analizador sintáctico el código que corresponde a la categoría detectada. También reconoce algunos errores sintácticos y elimina los comentarios.

Consideremos, para concretar, un lenguaje de programación cuyas categorías fueran:

Categorías	Códigos
- identificadores (etiquetas, variables..)	1
- números enteros	2
- delimitadores de 1 carácter: +,*,-,/,=	3,4,5,6,7
- delimitadores de 2 caracteres: **,/*,*/	8,9,10
- comentario	11

Los comentarios en este lenguaje pueden incluirse en cualquier parte del programa, delimitados entre /* y */: "/* - ESTO ES "/* ESTO ES UN COMENTARIO */".

Para simplificar, no consideremos limitación alguna en el número de caracteres de los identificadores ni de los números.

La gramática puede venir definida por:

E_A = {S,I,N,R1,R2,C,PFC}; E_T = {b,l,c,+,*,-,/,=,oc}

(abreviaturas: I = identificador ; b = blanco (espacio)
N = número ; l = letra
R1 = resto 1 ; c = cifra
R2 = resto 2 ; oc = otro carácter
C = comentario ;
PFC= posible fin comentario)

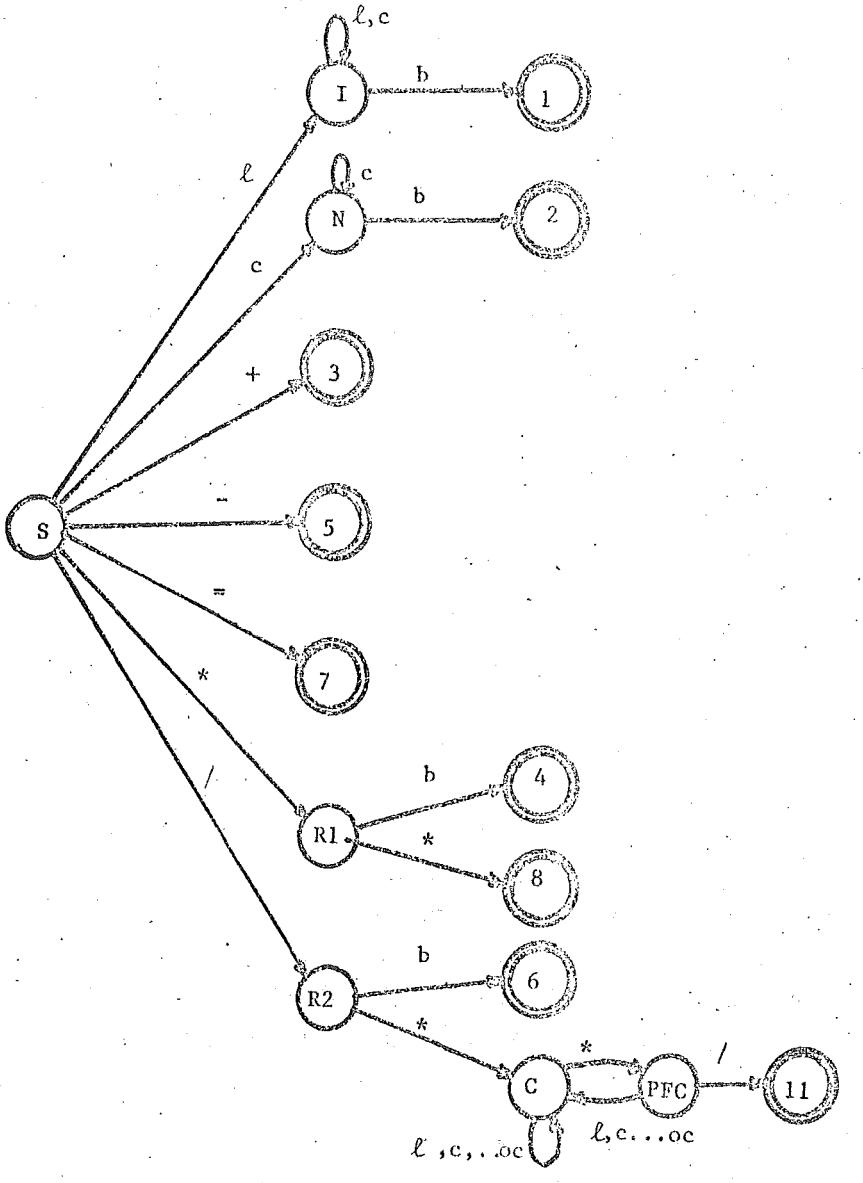
y las reglas:

S → lI|cN|+|-|=|*R1|/R2
I → b|lI|cI
N → b|cN
R1 → b|*
R2 → b|*C
C → * PFC|bC|lC|cC|+C|-C|/C|=C|ocC
PFC → /|bC|lC|cC|+C|-C|=C|ocC

Obsérvese que la gramática así definida es regular. Por tanto, existe un reconocedor finito del lenguaje. Su diagrama de Moore, de acuerdo con lo visto en el Capítulo anterior, será según la figura de la página siguiente. Aquí no hemos considerado un solo estado final (X), sino tantos como posibles categorías, y los hemos denominado con el código correspondiente.

Así, la modelación como reconocedor finito es una bue

na ayuda para programar el explorador. El analizador sintáctico, al llamarlo, lo inicializa en S y le entrega una cadena de caracteres. Si tal cadena es, por ejemplo, una sucesión de cifras y luego un blanco, entonces el reconocedor va al estado final 2, lo que quiere decir que el explorador devuelve el código 2 al analizador sintáctico.



El diagrama puede completarse con las transiciones que faltan desde cada estado, que conducirán a estados de error. - Por ejemplo, del estado N,con cualquier carácter que no sea c ni b se irá al estado final 2E, que significará que el explora dor devuelve control al analizador sintáctico indicándole que es un número incorrecto.

Señalemos que, aunque no hayamos hecho mención de ello, en los ensambladores también es necesario un explorador. Así, en el ejemplo del ensamblador de ENSAM, cuando en los organigramas tenemos "examinar tal campo", lo que se hace realmente es llamar a un subprograma explorador.

3.3. Análisis sintáctico.

Así como el explorador realiza un análisis a nivel de caracteres, el analizador sintáctico lo hace al nivel de sentencias. Aparte de algunas verificaciones preliminares (como, por ejemplo, comprobación de que hay el mismo número de paréntesis cerrados y abiertos),la parte fundamental del analizador sintáctico es un algoritmo llamado *reconocedor* ("parser"). Este algoritmo tiene como datos de entrada las reglas gramaticales del lenguaje que se está compilando y la cadena de categorías sintácticas entregada por el explorador,α; lo que debe - hacer es decidir si $S \overset{*}{\Rightarrow} \alpha$ es una derivación válida en esa - gramática. Sabemos que los lenguajes de programación pueden - representarse con gramáticas de contexto libre; no debe, pues, sorprender, que todos los algoritmos de reconocimiento utilicen una o varias *pilas* simuladas en memoria central.

Se han propuesto y se utilizan diversos algoritmos de reconocimiento. El más sencillo es el llamado *reconocedor descendente con retroceso*, en el que, partiendo del símbolo inicial, se aplican sucesivamente las reglas hasta obtener un árbol de derivación cuyo resultado sea la sentencia analizada,α.

Así, si las reglas iniciales son $S \to \alpha_1 | \alpha_2 \ldots | \alpha_n$, se comprobará primero si, aplicando las otras, resulta $\alpha_1 \overset{*}{\Longrightarrow} \alpha$; si no, se prueba con α_2, y así sucesivamente. Puede intuirse que el método exige una gran cantidad de ensayos y retrocesos, lo que lo hace bastante lento. Existen otros algoritmos que reducen el tiempo de reconocimiento: descendentes recursivos, ascendentes, etc.

3.4. Análisis semántico.

En la fase de análisis semántico se detecta la validez semántica de las sentencias aceptadas por el analizador sintáctico. Por ejemplo, la sentencia <variable> = <expresión> es semánticamente válida si ambos son de tipo compatible.

Cada regla de la gramática tiene asociada una subrutina semántica.

3.5. Generación del código objeto.

El programa se pasa a un formato intermedio. Si la ejecución es interpretativa, esta forma del programa será la entrada a un intérprete, que lo ejecutará. Si no es así, habrá que pasar a la versión en lenguaje de máquina o ensamblador, a la que se suele denominar *código objeto* o simplemente, *código*. En cualquier caso, el tipo de formato intermedio depende de la estructura de las instrucciones de máquina.

3.5.1. Cuartetos.

Se utiliza el formato de cuartetos cuando las instrucciones son de tres direcciones. El formato tiene cuatro campos:

<operador>, <operando 1>, <operando 2>, <resultado>.

Así, la sentencia A = B*C se representaría:

```
*,B,C,A
```

Y

R = (A + B*C)/D:

```
*,B,C,M1
+,A,M1,M2
/,M2,D,R
```

3.5.2. Tercetos.

El formato de tercetos es adecuado para instrucciones de máquina con dos direcciones; tiene tres campos:

<operador>, <operando 1>, <operando 2>

El resultado del terceto n se referencia por (n). Así, para el mismo ejemplo anterior tendremos:

```
1    *,B,C
2    +,A,(1)
3    /,(2),D
4    =,R,(3)
```

3.5.3. Notación polaca.

La notación polaca es muy útil cuando las instrucciones de máquina sólo tienen una dirección, y, de una manera general, para evitar ambiguedades. La notación que habitualmente empleamos es la notación *infijo*, en la cual las sentencias pueden considerarse derivadas de reglas del tipo

<expresión> → <operando 1> <operador> <operando 2>

(P. ej.: A + B). En notación polaca *sufijo* las reglas serían:

<expresión> → <operando 1> <operando 2> <operador>

(P. ej.: AB +).

Utilizando esta notación no es necesario utilizar paréntesis. Por ejemplo,

(A+B)*C en notación polaca sería AB+C*

A+(B*C) " " " " ABC*+

La notación polaca se puede generalizar para expresar todo tipo de sentencias. Por ejemplo, una sentencia de asignación, como "A=B", se escribirá: "AB="; una condicional, como "si A=B entonces C=A*B si no C=A/D" resultará: AB= si CAB*= entonces CAD/= si no".

Este tipo de notación se aproxima mucho a la forma de generación del código, ya que, recorriendo la cadena de izquierda a derecha, se encuentra la operación a efectuar cuando ya se conocen los dos operandos. Por ejemplo, si hay que generar el código correspondiente a

R = ((A/(B+C)) -D)*E,

que, en notación polaca, será

RABC+/D-E*=,

iremos explorando de izquierda a derecha hasta encontrar un operador; lo aplicamos a los operandos que le preceden, guardamos el resultado en un variable intermedia y empezamos de nuevo desde la izquierda. Suponiendo que el código a generar es ENSAM, tendremos

R A B C + / D - E * =	:	CAR	B
V		SUM	C
		ALM	V
R A V / D - E * =	:	CAR	A
W		DIV	V
		ALM	W

```
R W D - E * =          :   CAR   W
      X                    RES   D
                           ALM   X
R X E * =              :   CAR   X
      Y                    MUL   E
                           ALM   Y
R Y =                  :   CAR   Y
                           ALM   R
```

Obsérvese la gran redundancia del código resultante, - que, en definitiva, se traduciría por una ocupación de memoria mucho mayor y una ejecución mucho más lenta que si se hubiera escrito directamente en ensamblador. De ahí que en el diseño - de todo compilador sea muy importante aplicar técnicas de optimización que permiten abreviar el código y, en consecuencia, - ahorrar memoria y reducir el tiempo de ejecución.

4. INTÉRPRETES.

Un intérprete es un traductor que ejecuta las instrucciones del programa fuente conforme las va leyendo. Generalmente, para que el tiempo de ejecución no sea excesivo,hay una - primera parte que, funcionando igual que las primeras fases de un compilador, traduce todo el programa fuente a un formato interno (p. ej., versión del programa en notación polaca). El intérprete propiamente dicho trabaja sobre este formato interno, y consta, esencialmente, de dos partes:

- un conjunto de subprogramas, uno por cada tipo de - sentencia del lenguaje;

- un bucle de interpretación que realiza la decodificación de una sentencia del programa fuente, esta--

blece los operandos y el subprograma a aplicar y, al final de la ejecución, determina la siguiente sentencia a ejecutar.

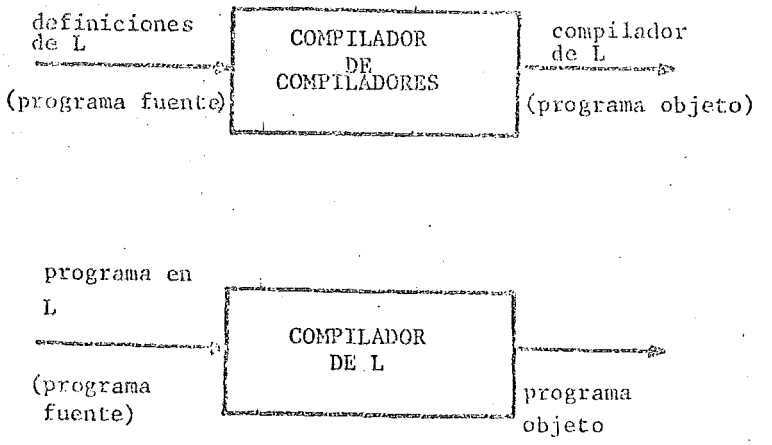
5. Sistemas de escritura de traductores. Compilador de compiladores.

El diseño y la programación de un compilador es un proceso lento y costoso que requiere la participación de un equipo de especialistas trabajando durante varios meses. Así, en los primeros compiladores de FORTRAN se invirtieron del orden de 18 hombres-año; actualmente, con el desarrollo de las técnicas de compilación, gracias a la formalización de los lenguajes, la escritura de un compilador puede necesitar unos 4 hombres-año.

Los sistemas de escritura de traductores son ayudas para automatizar algunas o varias fases de la construcción de un traductor y así reducir su coste y tiempo de desarrollo. Por ejemplo, un *constructor* es un programa que, recibiendo como entrada las reglas gramaticales del lenguaje, produce un algoritmo de reconocimiento.

El caso más típico y más completo de sistema de escritura de traductores es el *compilador de compiladores*. Se trata de un programa que acepta la descripción de un lenguaje de programación y produce un compilador para ese lenguaje. La entrada del compilador de compiladores será pues un programa escrito en un *metalenguaje* para describir L; la ejecución del compilador de compiladores se puede llamar *metacompilación*, y produce el compilador que luego aceptará y compilará programas escritos en el lenguaje L.

Normalmente, el compilador producido con un compilador



de compiladores es menos eficaz que si se hubiera programado en ensamblador tanto por ocupar más memoria como por ser más lento en ejecución. Sin embargo, parece que tenderán a utilizarse cada vez más, del mismo modo que se tiende a utilizar los lenguajes de alto nivel en lugar de los ensambladores.

6. NOTAS HISTÓRICA Y BIBLIOGRÁFICA.

El primer paso para abandonar la programación en lenguaje de máquina en favor de un lenguaje simbólico tuvo su origen en los trabajos llevados a cabo al comienzo de los años 50 en el M.I.T., que dieron como fruto el primer lenguaje de nivel ensamblador, el SAP (Symbolic Assembly Program), aparecido en 1.952, para el ordenador IBM 701.

El primer compilador fue el AO, desarrollado por Hopper, un investigador de Remington Rand, para el Univac 1. La versión preliminar del FORTRAN apareció en 1954 para el IBM 701, pero la versión definitiva de lo que hoy se conoce como FORTRAN I, y el primer manual, aparecieron en 1.956, elaborados por Backus, para el IBM 704.

Al comienzo de los años 60, y como consecuencia de los trabajos de Chomsky, surgen los primeros ensayos sobre compilación orientada por sintaxis, es decir, definir un lenguaje de programación como unas tablas de sintaxis y unas tablas indicadoras de las operaciones a realizar sobre las diferentes unidades sintácticas. Inmediatamente aparece la idea del compilador de compiladores.

La notación polaca debe su nombre a Luckasiewicz, matemático polaco que la introdujo.

Obra de consulta recomendable, traducida al español, es la de GRIES (1971).

REFERENCIAS BIBLIOGRAFICAS

ARBIB, M.A. Theories of abstract automata. Prentice-Hall, Englewood Cliffs, N.J., 1.969.

BACKUS,J.W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. Proc. Internat. Conf. Information Processing, - UNESCO, París, 1.959, 125-132.

BAR-HILLEL, Y., PERLES, M. y SHAMIR, E. On formal properties - of simple phrase structure grammars. Zeitschrift für - Phonetic, Sprachwissenschaft und Kommunikationsforschung, 14 (1.961), 143-172.

CANTOR, D.C. On the ambiguity problem of Backus systems. Journal of ACM, 9 (1.962), 4, 477-479.

CHOMSKY,N. Three models for the description of language. I.R.E. Trans. Inf. Theory, IT-2 (1.956), 113-114.

CHOMSKY,N. On certain formal properties of grammars. Information and Control, 2 (1.959), 137-167.

CHOMSKY,N. Context-free grammars and pushdown storage. Quart. Prog. Dept. No. 65 (1.962), MIT res. Lab. Elect., - 187-194.

CHOMSKY,N. Prólogo al libro de GROSS y LENTIN (1.967).

CHOMSKY,N. Language and mind. Harcourt Braze, New York,1.968.

CHOMSKY,N. The logical structure of linguistic theory. Plenum Press, New York, 1.975.

CHOMSKY,N. y MILLER, G.A. Finite state languages. Information and Control, 1 (1.958), 91-112.

CHOMSKY,N. y SCHUTZENBERGER,M.P. The algebraic theory of context-free languages. Computer Programming and Formal Systems. North-Holland, Amsterdam, 1.963, 118-161.

EVEY,J. The theory and application of pushdown store machines. (Tesis Doctoral). Harvard University, Cambridge, Mass., 1.963.

FLOYD,R.W. On ambiguity in phrase structure languages. Comm. of ACM, 5 (1.962), 10, 526-534.

FLOYD,R.W. The syntax of programming languages-A survey I.R.E. Trans. Electronic Computers, 14 (1.964), 4, 346-353.

GINSBURG,S. The mathematical theory of context-free languages. McGraw-Hill, New York, 1.966.

GINSBURG,S. Lectures on context-free languages. En ARBIB, M.A. (ed.): Algebraic theory of machines, languages, and semigroups. Academic Press, New York, 1.968.

GROSS,M. y LENTIN,A. Notions sur les grammaires formelles. Gautier-Villars, París, 1.967. (Edición española: Tecnos, Madrid, 1.976).

GRIES,D. Compiler construction for digital computers. Wiley, New York, 1.971 (Edición española: Construcción de compiladores. Paraninfo, Madrid, 1.974).

HARRISON,M.A. Introduction to schwitching and automata theory. McGraw-Hill, New York, 1.965.

HARRISON,M.A. Introduction to formal language theory Addison-Wesley, Reading, Mass., 1.978.

HOPCROFT,J.E. y ULLMAN,J.D. Formal languages and their relation to automata. Addison-Weslwy, Reading, Mass., 1969.

KURODA,S.Y. Classes of languages and linear-bounded automata. Information and control, 7 (1.964), 2, 114-125.

LURIA,A.R. Cerebro y lenguaje. Fontanella, Barcelona, 1.974a

LURIA,A.R. Cerebro en acción. Fontanella, Barcelona, 1.974b

MYHILL,J. Linear bounded automata. WADD Tech. Note 60-165, Wright Patterson Air Force Base, Ohio, 1.960.

NAUR,P. (Ed.) Revised Report on the algorithmic language ALGOL 60. Comm. of ACM, 6 (1.963), 1, 1-17.

OETTINGER,A.G. Automatic syntactic analysis and the pushdown store. Proc. Symp. Applied Math., Amer. Math. Soc., Providence, Rhode Island, 1.961.

RABIN,M.O. y SCOTT,D. Finite automata and their decision problems. IBM Journal Res. Dev., 3 (1.959),2, 114-125.